KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Block-quantized and data-efficient deep learning at the edge

Optimizing deployments with general-purpose and domain-specific vector instruction sets

Nitish Satya Murthy

Supervisors:
Prof. dr. ir. Marian Verhelst
Prof. dr. ir. Francky Catthoor

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

March 2025

Block-quantized and data-efficient deep learning at the edge

Optimizing deployments with general-purpose and domain-specific vector instruction sets

Nitish SATYA MURTHY

Examination committee:
Prof. dr. ir. Hendrik Van Brussel, chair
Prof. dr. ir. Marian Verhelst, supervisor
Prof. dr. ir. Francky Catthoor, supervisor
Prof. dr. ir. Toon Goedemé
Prof. dr. ir. Peter Karsmakers
Prof. dr. ir. Steven Latré
(Imec, Leuven & University of Antwerp)
Prof. dr. ir. Marina Zapater Sancho
(REDS institute, HES-SO, Switzerland)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

March 2025

© 2025 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Nitish Satya Murthy, Kasteelpark Arenberg 10, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Acknowledgements

This PhD journey has undoubtedly been one of the most challenging yet rewarding experiences of my life. The numerous hurdles and obstacles, starting with the initial pandemic phase, would have been impossible to overcome without the support of the incredible people around me.

First and foremost, my heartfelt thanks and appreciation go to my supervisors, Prof. Marian Verhelst and Prof. Francky Catthoor.

Marian, your passion, energy and enthusiasm for research and innovation have been a constant source of inspiration. You have always encouraged and motivated me to not shy away from setting ambitious goals. This support has empowered me to push beyond my limit and strive for continuous improvement. Your insights and feedback, both technical and non-technical, have been invaluable, and they will continue to guide me in becoming a better version of myself.

Francky, I feel privileged to have been one of your last PhD students. You have always encouraged me to see the bigger picture and work toward it, ensuring that I stayed on track despite my occasional deviations. Your guidance has been crucial in helping me plan and organize my research more effectively. Your extensive knowledge of psychology and the wisdom you've shared will continue to benefit me in the future.

I would have been grateful to be mentored by either one of you, but having both of your support throughout these years has been an invaluable privilege. Your efficiency, clarity of thought, remarkable reviewing speed, and immense work output among others often made me wonder whether you had more than 24 hours in a day or some magical powers. The only challenge was finding common meeting slots in your busy schedules, yet you always made time for our discussions. I deeply appreciate your patience with my planning and organizational chaos, including publication deadlines among others, and I truly value your steadfast support. I can only aspire to be like you, thank you for being such wonderful role models.

I would like to sincerely thank Prof. Hendrik Van Brussel for chairing the examination committee, and Prof. Toon Goedemé, Prof. Steven Latré, Prof. Peter Karsmakers and Prof. Marina Zapater Sancho for reading my thesis and providing valuable suggestions for improvement. I am also grateful to Prof. Goedemé and Prof. Latré for their periodic supervision, which helped me achieve critical milestones on time. Thanks also to Prof. Karsmakers and Prof. Zapater Sancho for agreeing to be part of my jury. Thank you to Prof. Zapater Sancho for traveling to attend my defence in person despite the constraints.

I extend my gratitude to Flanders AI and Imec for funding this research project. Thanks to Peter Debacker for his help in onboarding me to Imec and for the insightful discussions that launched my research. I am also grateful to Arindam for his constant support in ensuring I had the necessary resources for this pursuit. Thanks to Peter Vrancx for helping me navigate the rapidly evolving field of algorithmic research, for the brainstorming sessions, and for his valuable feedback on presentations and paper writing. Thanks to Timon for being a supportive and approachable manager, and always caring about my well-being during difficult times. Thanks to Paola for assisting with conference travels, and organizing engaging and enjoyable events. Thanks to Debjyoti for always being available for discussions and providing valuable feedback on my research. Thanks to Darayus for your constant encouragement and confidence-boosting conversations which have been incredibly helpful.

Thanks to Nathan for all the quantization discussions, for being an excellent sounding board to bounce ideas off, and for guiding me along a path you had already navigated, making my journey easier. I would pay to hear your floating point rants, and all other entertaining stories. Thanks to Giuseppe for your support throughout, the constructive feedback in achieving more ambitious goals, and the student supervision sessions through which I also learned a lot. Your discipline and dedication to work while balancing other activities have always been an inspiration.

Thanks to Emil for your encouragements throughout this journey. I enjoyed our bouldering sessions, the meme discussions, and look forward to our future celebration plans. Thanks to Diana for always being so kind and caring. Your planning and organization skills will always be inspirational. Thanks to Tommaso for always being available for gem5 or any other discussions. Thanks to Kalliopi for always being friendly, and keeping the office green and cheerful. Thanks to Giuliano for the fun weekend hangouts and for introducing me to all the pets I could meet through you. Thanks to Anmol, Diksha and others for making the ACACES summer school, with all the lectures, Al Dente pasta, even more enjoyable and memorable. Thanks to Bowen for always being ever-smiling and uplifting our moods during difficult times. Thanks to Leandro for the interesting and fun conversations while sitting next to you in the office. Thanks

to Josse and Joren for all the engaging compiler discussions. Thanks to Stef and Chao for all the block quantization conversations. Thanks to Ryan, for always being extremely cheerful and creating a happy atmosphere. Thanks to Nimish and Vikram for showing me all the nice restaurants in Leuven, especially for Indian food. Thanks to Linyan for always being so supportive and helpful. Thanks to Pengbo, Giovanni, Dwaipayan, and Prof. David Atienza, for the collaborations that have greatly helped improve my research.

Throughout this PhD journey, I had the opportunity to enjoy regular sporting sessions, keeping me active and providing a break from the computer screen. Thanks to Rico and Ryan for organizing all the badminton sessions regularly, and to Anmol, Darayus, Nektar, Kartik, Xiaoling, Amanda, Linyan, Nimish, Vikram, among others, for taking part and making it an enjoyable and relaxing weekend activity. Thanks to Erwan for organizing and planning the basketball sessions and to Abubakr, Giuseppe, Manu, Odysseas, Yukai for actively participating and making it a great stress-relieving evening exercise session.

Thank you to the rest of the CSA team for your support throughout: Arjun, Arnab, Bart, Benjamin, Boris, Diederik, Hadi, Jonas, Joyjit, Katayoon, Peter, Philipp, Saeideh, Satya, Sourav, Udari, Uras, Vinay, Weijiang, Wenzhe, Yunfan and others. Having a diverse research topic and being part of an interdisciplinary team has been immensely helpful.

Thanks also to the other MICAS and Marian's team members – Adrian, Arne, Fanchen, Guilherme, Jiacong, Jun, Pouya, Robin, Sander, Shirui, Yunhao, and others. The team meetings I attended were both interesting and informative. I also enjoyed the other fun activities during team get-togethers and outdoor trips. Thanks also to Ann, Anne and others at ESAT for all the administrative work that made my tasks easier.

Thank you also to my friends from RV PU, the Wing-4A group, the Berliner group and many others, for all the fun times and staying connected despite our geographical separation. This was especially nice during the lockdowns.

A big thank you to my family for their unwavering support and sacrifices that have made my endeavors possible. Born to parents who are both academics, I have been drawn to science from a young age. They continually motivated and encouraged me to pursue my interests and dreams, for which I am deeply grateful. My heartfelt thanks also go to my sister and brother-in-law for their constant support and belief. There were moments when my family had more faith in me than I had in myself, giving me the courage to face challenges without hesitation. You have stood by me through difficult times and celebrated my achievements, and I eagerly look forward to celebrating this milestone together.

Abstract

The demand for Artificial Intelligence (AI) continues to grow, leading to it becoming an integral aspect in our everyday lives. This surge can largely be attributed to the remarkable progress in Deep Learning (DL), which has consistently pushed performance boundaries often even outperforming human capabilities. Recent trends have focused on performance improvements through two major scaling axes: training data and model sizes. However, such scaling is often infeasible for practical DL on edge devices targeting custom and personalized tasks. This has driven research into data-efficient algorithms, which aim to deliver high task performance with limited data, addressing scenarios where traditional DL algorithms may falter.

While data-efficient algorithms tackle the challenge of data availability, they often overlook the stringent resource constraints inherent to edge devices. Additionally, model adaptation at the edge is often required to effectively handle data drift, distribution mismatches, etc., further complicating deployments. Block-quantization is used in this thesis to enable efficient computation at the edge, while maintaining algorithmic performance even at reduced-precision. The algorithmic landscape for these workloads is evolving rapidly, motivating the use of instruction-programmable processors that are resilient to obsolescence and also avoid high customization design costs. Moreover, improved performance and efficiency of deployments can be achieved by using vector instructions that are suitable for diverse data-parallel workloads predominantly found in DL.

This thesis firstly explores the data-efficient DL modality of few-shot image classification task, which overcomes the challenge of learning a new unseen task with very limited training data, as low as 1 or 5 samples per class. To enable efficient DNN adaptation on resource-constrained edge devices, block-quantization is used during the training process in both the forward and backward passes. It is also shown that conventional meta-learning algorithm initializations used for such tasks fail to transfer well for these quantized adaptations and can be improved by a HW-aware meta-learning algorithm that

upfront includes the target HW constraints for DNN adaptations.

In the next part, the focus is still on data-efficiency but targeting a different modality of continuous control tasks such as robotic control. The objective here is to attain effective task control while not requiring large number of environmental interactions. The usage of block-quantized DNNs as dynamics models is introduced here, leading to more efficient inference deployments, and in turn more HW-efficient planning in a conventional model-based RL algorithm. A systematic study of the impacts of quantization error on the degradation of control performance is performed, which typically exacerbates with any model prediction inconsistencies due to the effect of compounding error in its future predictions. Optimizing such a process while still maintaining effective control is achieved through flexible block configurations and utilizing higher precision only for more sensitive computations.

Next to these algorithmic impacts of block-quantization, the exploration of the effect of running them on instruction-programmable processors is also performed. Block-scaled GeMM kernels are designed and optimized across multiple design parameters, with only the use of general-purpose vector instructions. System-level analysis is done to benchmark these kernels against standard datatypes. The impact of increasing block sizes for the same workload leads to decreasing cycle counts due to lower exponent handling overhead. Similarly, 2D blocks are introduced as opposed to conventional 1D blocks, also providing increased performances. To reduce the cost of memory transfers, effective fused requantization has also been designed. These kernels were also evaluated for usage in a DNN training workload. A co-optimized process of adaptive training with vector and square block shapes is proposed to trade off GeMM computational speeds and accuracy.

Lastly, the deployments using domain-specific vector instructions were explored for block-quantized DNN computations with a software-defined SIMD HW unit. Such a HW unit enabled high flexibility for runtime variable-precision computations exhibiting different data-level parallelism, that can be effectively utilized for DNNs. Several mixed-precision inference deployments of CNNs were analyzed, providing several HW-SW trade-off optimization potential.

The contributions of this thesis help push the optimization boundaries of deployments of data-efficient DL algorithms at the edge, using general to domain-specific vector instruction sets, with the use of block-quantization techniques. The progress of research in this interdisciplinary intersection is crucial to enable futuristic applications of bringing intelligence to everyday devices in a performant and privacy-driven manner.

Beknopte samenvatting

De vraag naar Kunstmatige Intelligentie (AI) blijft groeien, waardoor het een integraal onderdeel van ons dagelijks leven wordt. Deze toename is grotendeels toe te schrijven aan de opmerkelijke vooruitgang in Deep Learning (DL), die consequent de prestatielimieten heeft verlegd en vaak zelfs menselijke capaciteiten overtreft. Recente trends richten zich op prestatieverbeteringen langs twee belangrijke schaalassen: trainingsdata en modelgroottes. Echter, dergelijke schaalvergroting is vaak onpraktisch voor DL-toepassingen op randapparaten die gericht zijn op aangepaste en gepersonaliseerde taken. Dit heeft onderzoek gestimuleerd naar data-efficiënte algoritmen die hoge taakprestaties kunnen leveren met beperkte data, gericht op scenario's waarin traditionele DL-algoritmen tekortschieten.

Hoewel data-efficiënte algoritmen de uitdaging van beperkte beschikbaarheid van data aanpakken, negeren ze vaak de strikte resourcebeperkingen die inherent zijn aan randapparaten. Daarnaast is modelaanpassing aan de rand vaak vereist om effectief om te gaan met datadrift, distributieverschillen, enz., wat implementaties verder bemoeilijkt. In dit proefschrift wordt blokkwantisatie gebruikt om efficiënte berekeningen aan de rand mogelijk te maken, terwijl de algoritmische prestaties behouden blijven, zelfs bij verminderde precisie. Het algoritmische landschap voor deze taken evolueert snel, wat het gebruik van instructie-programmeerbare processors motiveert die bestand zijn tegen veroudering en tevens hoge ontwerpkosten voor maatwerk vermijden. Bovendien kunnen verbeterde prestaties en efficiëntere implementaties worden bereikt door gebruik te maken van vectorinstructies die geschikt zijn voor uiteenlopende data-parallelle taken, zoals die vaak voorkomen in DL.

Dit proefschrift onderzoekt eerst de data-efficiënte DL-modaliteit van few-shot beeldclassificatietaken, waarmee de uitdaging wordt overwonnen om een nieuwe, onbekende taak te leren met zeer beperkte trainingsdata, soms slechts 1 of 5 voorbeelden per klasse. Om efficiënte DNN-aanpassing op resourcebeperkte randapparaten mogelijk te maken, wordt blokkwantisatie toegepast tijdens het

trainingsproces in zowel de voorwaartse als achterwaartse passes. Er wordt ook aangetoond dat conventionele meta-learning algoritme-initialisaties voor dergelijke taken slecht overdraagbaar zijn naar deze gekwantiseerde aanpassingen en verbeterd kunnen worden door een HW-bewust meta-learning algoritme dat de hardwarebeperkingen vooraf integreert in de DNN-aanpassingen.

In het volgende deel blijft de focus op data-efficiëntie, maar wordt een andere modaliteit onderzocht: continue regeltaken, zoals robotbesturing. Het doel hier is om effectieve taakcontrole te bereiken zonder een groot aantal interacties met de omgeving. De toepassing van blok-gekwantiseerde DNNs als dynamische modellen wordt geïntroduceerd, wat leidt tot efficiëntere inferentie-implementaties en daardoor een meer HW-efficiënte planningsaanpak binnen een conventioneel model-gebaseerd RL-algoritme. Er wordt een systematische studie uitgevoerd naar de impact van kwantisatiefouten op de afname van de regelprestaties, wat doorgaans verergert door inconsistenties in modelvoorspellingen als gevolg van de cumulatieve fout in toekomstige voorspellingen. Het optimaliseren van dit proces terwijl effectieve controle behouden blijft, wordt bereikt door flexibele blokconfiguraties en het gebruik van hogere precisie alleen voor de meest kritische berekeningen.

Naast deze algoritmische effecten van blokkwantisatie wordt ook het effect van de uitvoering ervan op instructie-programmeerbare processors onderzocht. Blok-geschaalde GeMM-kernen worden ontworpen en geoptimaliseerd over meerdere ontwerpparameters, uitsluitend met behulp van generieke vectorinstructies. Een systeemanalyse wordt uitgevoerd om deze kernen te benchmarken tegen standaard datatypes. Het vergroten van blokgroottes voor dezelfde taak leidt tot een afname van het aantal klokcycli door lagere overhead bij exponentverwerking. Op dezelfde manier worden 2D-blokken geïntroduceerd in plaats van conventionele 1D-blokken, wat eveneens tot prestatieverbeteringen leidt. Om de kosten van geheugentransfers te verminderen, is een effectieve gefuseerde herkwantisatie ontworpen. Deze kernen worden ook geëvalueerd voor gebruik in een DNN-trainingsworkload. Een gezamenlijk geoptimaliseerd proces van adaptieve training met vector- en vierkante blokvormen wordt voorgesteld om een afweging te maken tussen GeMM-berekeningssnelheden en nauwkeurigheid.

Tot slot wordt de implementatie van domeinspecifieke vectorinstructies onderzocht voor blok-gekwantiseerde DNN-berekeningen met een software-gedefinieerde SIMD HW-eenheid. Een dergelijke HW-eenheid biedt hoge flexibiliteit voor runtime variabele-precisie berekeningen met verschillende vormen van dataparallelisme, die effectief kunnen worden benut voor DNNs. Verschillende gemengde-precisie inferentie-implementaties van CNNs worden geanalyseerd, wat meerdere HW-SW optimalisatie-afwegingen mogelijk maakt.

De bijdragen van dit proefschrift helpen de optimalisatiegrenzen te verleggen voor de implementatie van data-efficiënte DL-algoritmen aan de rand, met behulp van generieke tot domeinspecifieke vectorinstructiesets en blokkwantisatietechnieken. Vooruitgang in dit interdisciplinair onderzoeksgebied is essentieel om toekomstige toepassingen te realiseren waarin intelligentie op een prestatiegerichte en privacy-bewuste manier naar alledaagse apparaten wordt gebracht.

List of Abbreviations

AI Artificial Intelligence. v, vii, 1, 3, 6, 7, 15, 23, 30, 76, 138

ARM Advanced RISC Machine. xx, xxi, xxv, 6, 9, 12, 35–37, 76, 77, 79–81, 85, 86, 88, 94, 95, 106, 108, 109, 111, 113, 116, 119–121, 125, 141, 146

ASIC Application-Specific Integrated Circuit. 5, 9, 34, 35, 75, 108

ASIP Application-Specific Instruction-set Processor. 5, 34

AWS Amazon Web Services. 2

BFP Block Floating Point representation. xix, xx, xxv, xxvi, 7, 30, 32–34, 39, 41, 44, 46–49, 53, 60, 61, 63–65, 67–74, 76, 103, 123, 140, 144, 149, 150

BS-INT Block-scaled integers. xxii, 76, 81, 103, 105, 123, 124, 126–137, 142, 143

BS-INT8 Block-scaled 8-bit integer representation. xx–xxiii, 77–79, 85, 87–91, 93, 95–105, 110, 112–116, 118–120, 141, 142, 151, 152

CEM Cross-Entropy Method. 61, 62, 66, 67, 69, 144

CNN Convolutional Neural Network. vi, viii, 13, 22, 23, 80, 96, 103, 105, 124, 127, 131, 135, 141, 143, 147

CPU Central Processing Unit. 5, 10, 34, 35, 80

DFA Direct Feedback Alignment. 26

DL Deep Learning. v–vii, ix, xix, 1–5, 7, 8, 19, 22–24, 35, 38, 56, 63, 124, 138, 144, 148

DLP Data-Level Parallelism. 34, 36

DNN Deep Neural Network. v–viii, xix, xx, xxii, 3–5, 7–13, 15–38, 40–42, 44, 46, 57–63, 65, 66, 69, 75–81, 83–85, 105–109, 111, 113, 115–117, 119–124, 128, 136–138, 140–142, 144–147

DRTP Direct Random Target Projection. 26

DSIP Domain-Specific Instruction-set Processor. 10, 12

DSP Digital Signal Processing. 7, 10, 32

FA Feedback Alignment. 26

FC Fully Connected. 12, 15, 17, 18, 20, 22, 108, 115, 116, 119, 142, 146

FP Floating-Point representation. 7, 9, 10, 28–30, 32, 33, 35, 36, 44, 54, 63, 68, 80, 81, 83, 91, 103, 109, 123–125, 145, 150

FP16 IEEE-754 16-bit Floating-Point representation. 9

FP32 IEEE-754 32-bit Floating-Point representation. xix–xxii, xxv, xxvi, 9, 12, 21, 28, 29, 35, 43, 45, 46, 48–50, 52, 53, 61, 65, 68–74, 76, 79, 80, 83, 87, 91, 95–100, 102, 103, 105, 108, 117–120, 123, 124, 126, 129, 130, 132–137, 140–143, 150–152

FPGA Field Programmable Gate Array. 5, 34, 35

FSL Few-Shot Learning. 7, 8, 39–42, 47, 49, 50, 54, 55, 139, 149

FXP Fixed-Point representation. 7, 28–30, 32, 33, 44, 80, 81, 83, 123

GeMM General Matrix Multiply. vi, viii, xx–xxii, xxv, 9, 12, 17, 18, 20, 39, 45, 47, 63–65, 67, 76–81, 83–91, 93, 95–98, 100–110, 112–114, 116, 118–120, 140–142, 145–147, 152

GeMV General Matrix-Vector Multiply. 17

GPT Generative Pre-trained Transformer. 1

GPU Graphics Processing Unit. 1, 5, 6, 34, 35, 47, 67

HW Hardware. v, vi, viii, xix, xxii, 3, 5, 6, 8–10, 12, 24, 25, 27–32, 34–39, 57, 58, 63, 68, 71, 73–77, 79–81, 83, 85, 93, 103–107, 109, 113, 116, 119–121, 123, 125, 126, 129–134, 138, 140–147, 149, 151

IMMA Integer Matrix-Multiply and Accumulate. 6

INT Integer representation. 76, 81, 102, 133

INT8 8-bit Integer representation. xxi, xxii, xxv, 9, 12, 31, 35, 70, 71, 74, 79, 81, 83, 84, 88, 94–100, 102, 103, 109, 112, 117–120, 122, 124, 126, 133–137, 141, 143

ISA Instruction Set Architecture. 9, 34, 36, 76, 79, 120, 121, 146

L2L Meta-learning (Learn to learn). xix, xx, xxv, xxvi, 46–53, 149

MAC Multiply-accumulate operation. xxi, xxii, 10, 17, 22, 23, 61, 63, 66, 68, 69, 73, 74, 76, 77, 83, 85–87, 93, 96–101, 103, 104, 122, 124, 128, 134, 135, 137, 141, 143, 144, 146, 151

MAML Model Agnostic Meta-Learning. 42, 55

MBRL Model-Based Reinforcement Learning. xx, 8, 11, 57–61, 63, 66, 69, 73, 74, 107, 140, 142, 144, 145

MCU Microcontroller Units. 3, 36

MFRL Model-Free Reinforcement Learning. xx, 8, 57–61, 63, 145

MLP Multi-layer perceptron. xix, 15–17, 22, 23

MPC Model Predictive Control. 59, 61, 62, 66, 140, 145

MSE Mean Squared Error. 19, 31, 72

MX Microscaling. 7, 10, 30, 33, 80, 138, 144, 146

NAS Neural Architecture Search. 5, 22, 23, 26, 27

NLL Negative Log-Likelihood. 19

NN Neural Network. xix, 15, 16, 22, 23, 53–55, 108, 115, 116, 119, 142, 146

OOE Out-of-order execution. 37

PETS Probabilistic Ensembles with Trajectory Sampling. xx, 61, 62, 68, 73, 140

PTQ Post Training Quantization. 31, 45, 63, 65

QAT Quantization Aware Training. 32, 33, 45, 63

qL2L Quantized Meta-learning. xix, xx, xxv, xxvi, 46–48, 50–53, 149

ReLU Rectified Linear Units. xix, 16, 18, 22, 47, 66, 68, 73, 116, 140

RL Reinforcement Learning. vi, viii, 8, 11, 12, 27, 56–59, 61, 63, 73, 140, 147

RVV Risc-V Vector instruction-set. 9, 36, 37, 77, 146

SGD Stochastic Gradient Descent. 20, 21

SIMD Single Instruction Multiple Data. vi, viii, xxii, 9, 12, 13, 34–37, 76, 79, 103, 123–128, 131, 136, 141–143, 146

SoTA State of the art. 42, 63

STE Straight Through Estimator. 32

SVE Scalable Vector Extension. xx, xxi, 6, 9, 12, 35–37, 76, 77, 79, 80, 85, 86, 88, 94, 95, 97, 98, 101, 102, 104, 106, 108, 109, 111, 113, 116, 119–121, 125, 141, 146

SW Software. vi, viii, xxii, 9, 10, 12, 27, 29, 30, 32, 34, 36, 75, 77, 83, 85, 106, 113, 119–121, 123, 138, 145–147

ViT Vision Transformer. 1

VL Vector register length. xxi, 85, 86, 95

VLA Vector Length Agnostic. xx, 9, 36, 37, 76–80, 85, 86, 99, 141, 146

WAGE Weight, Activation, Gradient and Error. xix, 39, 48, 54, 140, 149, 150

Contents

Abstract v

Beknopte samenvatting vii

List of Abbreviations xiv

Contents xv

List of Figures xix

List of Tables xxv

1 Introduction 1
- 1.1 The deep learning era . . . 1
  - 1.1.1 Data-efficient deep learning . . . 3
  - 1.1.2 Hardware-efficient deep learning . . . 5
- 1.2 Open research questions . . . 6
- 1.3 Thesis organization and contributions . . . 10

2 Deep neural network 15
- 2.1 DNN inference and training . . . 16
  - 2.1.1 Forward propagation . . . 16
  - 2.1.2 Loss calculation . . . 19
  - 2.1.3 Backward propagation . . . 19
  - 2.1.4 Weight update . . . 20
- 2.2 Common DNN architectures . . . 21
- 2.3 Conclusion . . . 23

3 Hardware-efficient DL 25
- 3.1 Efficient DNN algorithms . . . 25
- 3.2 Quantization . . . 28

3.2.1 Different datatypes for DNNs . . . 28
3.2.2 Quantization methodology . . . 30
3.3 Block-scaled data formats . . . 32
3.4 Programmable vector instruction set processors for DNNs . . . 34
3.5 Conclusion . . . 38

4 Hardware-aware meta-learning for quantized few-shot learning 39
4.1 Introduction . . . 40
4.2 Background theory . . . 42
4.2.1 Learn to learn - Meta-learning (L2L) . . . 42
4.2.2 Learn on chip - Quantized training . . . 44
4.3 Learn to learn on chip - Quantized meta-learning (qL2L) . . . 45
4.4 Experimental setup and results . . . 47
4.5 Analysis . . . 51
4.5.1 Effect of hyperparameters during quantized meta-learning 51
4.5.2 Effect of different fine-tuning with same initializations . 52
4.6 Related work . . . 54
4.7 Conclusions . . . 56

5 Hardware-efficient planning in model-based RL for continuous control tasks 57
5.1 Introduction . . . 59
5.2 Background theory on MBRL algorithm . . . 61
5.3 Related work . . . 63
5.4 Block-quantized dynamics model for MBRL . . . 63
5.5 Experimental setup . . . 65
5.5.1 Tasks . . . 66
5.5.2 Metrics of evaluation . . . 66
5.5.3 Simulation setup . . . 67
5.6 Results and analysis . . . 68
5.6.1 Impacts of DNN activation function . . . 68
5.6.2 Impacts of different quantization configurations for DNN dynamics model . . . 68
5.6.3 Impacts of mantissa bit-width and block sizes . . . 71
5.6.4 Quantization analysis for halfcheetah tasks . . . 72
5.7 Conclusion . . . 73

6 Block-scaled GeMM deployments using general-purpose vector instructions 75
6.1 Introduction . . . 76
6.2 Background and related work . . . 79
6.3 Methods . . . 85
6.3.1 Loop unrolling on vector processors . . . 86

6.3.2 Block-scaled integer GeMMs . . . 87
6.3.3 Flexible block granularities - 1D vector vs 2D blocks . . 90
6.3.4 Requantization methodologies - static vs dynamic . . . 91
6.4 Experimental setup . . . 94
6.5 Results and analysis . . . 95
6.5.1 Baseline GeMM performances . . . 96
6.5.2 BS-INT8 GeMM kernel performance comparisons . . . 97
6.5.3 Optimizing BS-INT8 GeMMs through activation block granularity . . . 99
6.5.4 Analyzing GeMM performances for different matrix dimensions . . . 100
6.5.5 Static and dynamic requantization strategies for BS-INT8 GeMMs . . . 100
6.5.6 Evaluation on CNN inference workloads . . . 103
6.6 Conclusion . . . 103

7 Block-scaled GeMM with adaptive block shapes for DNN training at the edge 106
7.1 Introduction . . . 107
7.2 Background and related work . . . 108
7.2.1 DNN training at the edge . . . 108
7.3 Block-scaled integer GeMMs for DNN training . . . 110
7.3.1 Optimizing block shapes for DNN training . . . 110
7.3.2 Vectorized BS-INT8 GeMM kernels for efficient computation on vector processors . . . 113
7.3.3 Adaptive DNN training through vectorized BS-INT8 GeMMs (vector and square blocks) . . . 113
7.4 Experimental setup . . . 115
7.5 Results and analysis . . . 117
7.5.1 DNN training convergence due to block-scaled datatypes 117
7.5.2 Benchmarking BS-INT8 vector and square GeMM kernels 118
7.5.3 Adaptive DNN training through vectorized BS-INT8 GeMMs (vector and square) . . . 119
7.6 Conclusion . . . 119

8 Domain-specific instruction-set processors towards more efficient block-scaled GeMM computation 122
8.1 Introduction . . . 123
8.2 Background . . . 125
8.2.1 Software-defined SIMD . . . 125
8.3 BS-INT computing microarchitecture . . . 126
8.3.1 Microarchitecture overview . . . 126
8.3.2 BS-INT arithmetic operation . . . 127

8.4 Algorithms quantization and mapping . . . 129
8.4.1 BS-INT quantization for inference . . . 129
8.4.2 Mapping to BS-INT . . . 130
8.5 Experimental setup . . . 131
8.5.1 Selected benchmarks and accuracy evaluations . . . 131
8.5.2 Hardware implementation . . . 133
8.5.3 Baseline setting . . . 133
8.6 Results . . . 134
8.6.1 Area performance . . . 134
8.6.2 Energy performance . . . 135
8.6.3 Inference performance . . . 135
8.7 Conclusion . . . 136

9 Conclusion and future work **138**
9.1 Thesis contributions . . . 139
9.2 Future work suggestions . . . 143

A Hyperparameter settings for few-shot learning experiments **149**

B Block-scaled DNNs **151**

Bibliography **155**

List of publications **181**

List of Figures

1.1 DL and its diverse application ranges. 2
1.2 DL model and dataset scaling 3
1.3 (a) and (b) Data variation at the edge. (c) Deep learning deployment configurations. 4
1.4 Overview of thesis contributions. 11

2.1 Structure of a NN / MLP with 6 inputs, 4 outputs and 3 hidden layers (8 neurons each). Activation function depicted here is a ReLU function. 16
2.2 Common DNN operations. 18

3.1 Taxonomy of efficient DNN algorithms. Taken from [11]. 26
3.2 DNN datatypes - conventional and block-scaled formats. 29
3.3 Overview of DNN processor platforms. With y-axis representing speed / performance, x-axis denoting the power consumption, and diagonal lines marking energy efficiencies. Taken from [75]. 34
3.4 SoftSIMD vs HardSIMD HW. [240] 38

4.1 Comparison between conventional meta-learning and our modified hardware-aware meta-learning process. 41
4.2 Quantized training by including fake quantizer nodes in forward and backward propagation [224] [247]. Quantizer nodes simulating BFP (x-bit). 44
4.3 Methodology for performing quantized meta-learning (qL2L) and the difference with conventional meta-learning (L2L). 46
4.4 Asymptotic accuracies after fine-tuning with different BFP quantizations using L2L initialization. (WAGE denotes Weights, Activations, Gradients and Errors. f represents FP32). 48
4.5 Fine-tuning curves for Omniglot classification. 49
4.6 Fine-tuning plots for qL2L (5-way, 5-shot). 52

4.7 Performance comparisons for different inner loop iterations in L2L and qL2L. 52
4.8 Fine-tuning loss curves for different outer loop iterations in qL2L. 53
4.9 Fine-tuning plots with all combinations of L2L/qL2L initializations and FP32/BFP quantized fine-tuning (5-way, 5-shot). . . 53

5.1 Conceptual role of DNNs for continuous control tasks and the general trade-offs among MBRL and MFRL. 58
5.2 Timing details of MBRL algorithms with environment interactions involving N trials with each trial having T timesteps of task horizon. Each timestep within the task horizon requires multiple sequential DNN inferences to decide the optimal action. 59
5.3 Methodology of trajectory sampling in PETS. With a state-action pair, the probabilistic ensemble network (PE) predicts the next state distributions. With initial state and an action sequence, the trajectories for each model in the ensemble are obtained by sequentially sampling states from the distributions. (Figure adapted from [38]) 62
5.4 Illustration of block floating point based GeMMs with different quantization options. "V" and "S" denote vector and square blocks with a common 8-bit exponent respectively. 64
5.5 Probabilistic ensemble networks with simulated BFP GeMMs (Current network architecture shown for the cartpole task). . . 65
5.6 Continuous control tasks along with their state and action dimensions. [215] 66
5.7 Impacts of different quantization datatypes and mixed-precision configurations of dynamics model on task returns. Bar plots show the mean and standard deviation of reward performance among 5 different runs. 69
5.8 Impacts of BFP mantissa bit-widths, block shapes and sizes. . . 71
5.9 Quantized model controller predictions for half-cheetah task. . 72

6.1 (a) Co-optimizations of deploying VLA programming models of DNNs on scalable vector processors. (b) Description of our VLA programming model for deployment of BS-INT8 GeMMs. . . . 78
6.2 ARM SVE includes 32 vector registers of scalable length 128xLEN bits (Z) and 16 predicate registers of length 16*LEN bits (P) (1 bit for the smallest controllable granularity of 8-bit lanes in a vector register). [204] 80
6.3 Common number representations in conventional and block-scaled formats. 81

6.4 (a) GeMM (A x W = O) computation examples using different ARM SVE MAC instructions spatially unrolled along VL. (All dimensions are in bytes unless specified otherwise). (b) Mapping onto vector registers–Z, using ARM SVE vector instructions. . 85
6.5 (a) BS-INT8 GeMM with separated integer and exponent parts and requantization for next layer at the register level without much data transfers and include in the microkernel. (b) Illustration of microkernel accumulations in BS-INT8 format with alignments at block boundaries (Block size - B). 89
6.6 Flexible granularity of activation blocks in the computation of BS-INT8 GeMM. Dotted lines represent microkernel accumulation dimensions. 90
6.7 Illustration of fused and dynamic requantization in vector processors through block-size based predication. 91
6.8 Design guideline for selecting kernel parameters based on workload requirements. 94
6.9 (a) Relative speedups of instruction / cycle counts with different MAC instructions. (b) Instruction count and (c) Cycle counts for INT8 and BS-INT8 GeMMs. 98
6.10 Relative speedups of cycle counts for BS-INT8 (*dp*) vs FP32 and INT8 (*dp*) with different granularities of activation blocks (1D and 2D). 99
6.11 Impacts of scaling along different GeMM dimensions evaluated on a 256-bit SVE vector length configuration. Plotting performance metric of MACs per cycles demonstrates the computational efficiency. 101
6.12 BS-INT8 requantization cycle-count speedup comparisons (All INT8 and BS-INT8 microkernels are *dp* type). Specific combinations of block size and vector lengths that enable fused operation are considered. Normalizations are with FP32 SVE 128 bits cycle counts. 102
6.13 BS-INT8 GeMM kernel profiling for running ResNet20 CNN models and showing speedup comparison against other baseline datatypes 104
6.14 BS-INT quantization impacts on CNN test accuracies. Analyzed different quantization bit-widths and activation block shapes (1D and 2D). 105

7.1 Block-scaled datatypes with 1D vector blocks (row/column) are demonstrated. 110

7.2 (a) Illustration of matrix multiplication microkernel (8x8) with integer parts and exponent parts (V8 and S8 configurations). (b) The optimizations in DNN training computation graph after reversing the orders of quantization and transpose operators in backward pass for square shapes. 111
7.3 Investigating commutative property of transpose and quantization operators for different block-scaled quantizations. 112
7.4 Pseudocode for our GeMM kernel using BS-INT8 formats with square and vector shapes. 114
7.5 Adaptive training for accuracy-speed trade-off. 115
7.6 Impacts of block-scaled datatypes (shape / sizes) on DNN training convergence (Baseline models – FP32 and INT8 (tensor-level scale)). 117
7.7 Vectorized GeMM kernel execution profiling for cartpole FCNN model with different datatypes depicting instruction/cycle counts, speed-up and memory compression ratios. 118
7.8 HW-SW trade-offs through our various BS-INT8 training configurations (in comparison to baseline FP32 and INT8 models) (Top figure). Training curves for different adaptive training ratios is also showed for the reacher task here (Bottom figure). 120

8.1 SoftSIMD (a) addition or subtraction, (b) shift. 125
8.2 SoftSIMD microarchitecture overview. 126
8.3 SoftSIMD arithmetic with BS-INT (a) sum of two multiplications, (b) shift-add-based multiplication, (c) addition between two BS-INT operands. 128
8.4 Example of accumulation precision increase during addition with a 48-bit datapath. Output sum is stored in 3 registers, and one of the operands is shifted before addition based on the difference between the exponent values. 129
8.5 Map BS-INT with SoftSIMD with different block granularities: (a) per-channel blocks and (b) multiple rows/channels. 131
8.6 (a) Test accuracies for mixed-precision quantized models with respective baseline FP32 accuracies (dotted lines), along with the impacts of different block granularities for activations. (b) The distribution density of accumulation shifts for ResNet20 and VGG16 models (6-bit activation and 4-bit weights with per-channel configuration). 132
8.7 Normalized per MAC (a) area and (b) energy. 134
8.8 Normalized TOPS/W and accuracy for (a) different BS-INT schemes with fixed activation-weight precision of 6-4 bits and (b) different mixed-precision configurations with per-channel block granularity. 136

9.1 Overview of thesis contributions. 139

B.1 Temporal accumulation choices for BS-INT8 datatypes. 152

List of Tables

3.1 Comparison of different datatype metrics. [60] 30
3.2 Energy and area costs in 45nm technology node. Taken from [66, 87]. 36
3.3 ARM-SVE supported processors. 37

4.1 Different experimental settings for analyzing quantized meta-learning. 48
4.2 Fine-tuning comparison of Omniglot (L2L vs qL2L initializations). 50
4.3 Different approaches for NN adaptation on hardware. 54

5.1 Hyperparameter settings and no. of MACs for PETS dynamics models (FP32). 67
5.2 Accumulated rewards for 5 random runs of each task (Tabular data shows: Mean ± std. across 5 runs). Higher rewards indicate better control performance. 68
5.3 Deployment and memory costs for PETS (FP32 vs BFP8). . . 70
5.4 INT8 model rewards comparison. 71
5.5 Accumulated rewards using quantized swish networks for the half-cheetah task. 73

6.1 ARM Vector Instructions used for the implementation of block-scaled GeMM kernels. 81
6.2 Block-scale based computing processors for DNNs. 82
6.3 Conceptual and implementation differences of our block-scaled GeMMs with BLIS algorithm. 84
6.4 MAC instructions with the microkernel accumulation steps and dimension. VL represents the specified vector length in bytes. . 87
6.5 Gem5 simulation setup details. 95

6.6 Instruction count splits and vectorization percentages for dp (INT8) and FP32 GeMM kernels of (128x128) x (128x128) matrix sizes. 97

A.1 Hyperparameter settings for Reptile meta-learning [152] (L2L) and quantized meta-learning (qL2L). f represents 32-bit floating point numbers, while others represent BFP mantissa-bitwidths. 149
A.2 Hyperparameter settings for conventional FP32 and BFP quantized fine-tuning 150

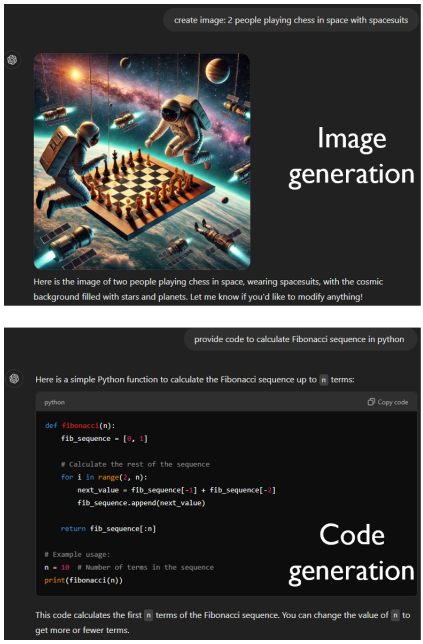
Chapter 1

Introduction

1.1 The deep learning era

The ever-increasing popularity of artificial intelligence (AI) and its widespread applications have made it an integral part of our life. More specifically, Deep Learning (DL) which is a subset of AI has been the main contributor to this growing interest. This has been achieved through effective algorithms, the increased availability of data in this data-driven age and powerful compute resources such as Graphics Processing Units (GPUs) [109]. Figure 1.1 illustrates a small sample of the diverse application scenarios of DL spanning from large-scale server applications like ChatGPT to small-scale personalized edge devices for vision and audio tasks. Figure 1.1a showcases the capabilities of a multi-modal universal chat-bot that responds to any queries, and Fig. 1.1b depicts smart devices that process data locally and elevate our daily life experiences.

The majority of DL models are trained by the "backpropagation" algorithm [188], allowing us to tune / train the black-box DL models for target datasets. While the same algorithm powers the advanced capabilities of DL across diverse tasks, several differences exist like the model size representing learning capacity and the number of training samples used to train the model. In recent years, as shown in Fig. 1.2a, we have observed trends of increased model sizes demonstrating improved performances across different domains such as image classification models (87 Million parameter model – ViT in 2021) [50], Large language models (175 Billion parameter model – GPT-3 in 2020) [19], etc. This improvement through scaling has been possible due to increased data availability as illustrated in Fig. 1.2b. ChatGPT models that have recently



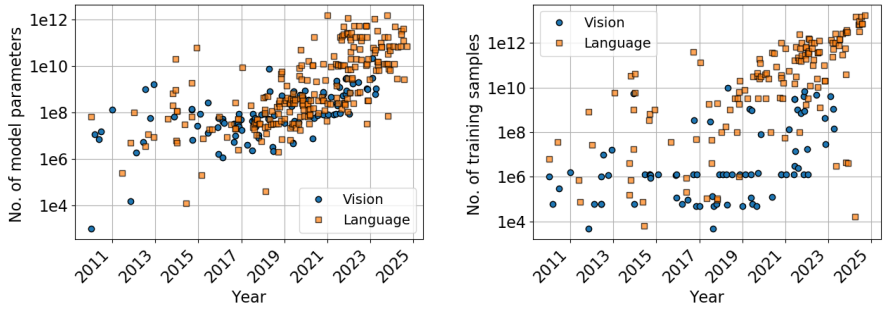
(a) ChatGPT (data-center) [26]



(b) Smart devices (edge) [71, 196]

Figure 1.1: Deep Learning (DL) and its diverse application ranges. (a) ChatGPT models running in server / data-centers: outputs based on user prompts – "create image of 2 people playing chess in space with spacesuits" (top) and "provide code to calculate Fibonacci sequence in python" (bottom). (b) Smart devices at the edge like AWS DeepLens camera [196] that can perform tasks like image classification using deep learning (top), Google Home speakers [71] that activate based on a wake word and further responds to your queries (bottom).

gained a lot of traction with their impressive capabilities, are estimated to be approximately 1.8 trillion parameter models and are trained with around 300 billion training samples/words [89]. However, there have been recent concerns regarding achieving further improvements through model scaling due to the lack of human data [221]. Additionally, there are concerns of sustainability and economic feasibility with growing energy demands to run these large models, with projected models needing to be powered by nuclear power plants [90]. This highlights the importance of also exploring data-efficiency for DL – both using small-sized but smart data, and investigating algorithms that perform well even with less data [111, 150].



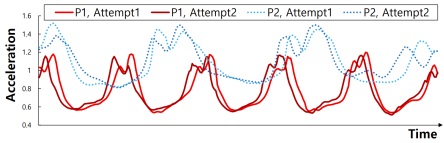
(a) **(b)**

Figure 1.2: Deep Learning (DL) scaling through the number of trainable parameters and the number of training samples of vision/language models [55].

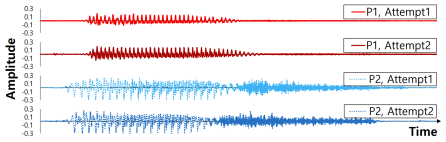
While DL improvement through scaling models and datasets could still be relied upon for cloud-based applications, they are not very feasible at the edge. Edge devices such as smartphones, embedded devices, wearables, tiny microcontrollers (MCUs), etc., often have limited compute power, smaller energy budgets (battery-operated), reduced storage capacity compared to compute resources at the cloud. Additionally, data at the edge is much more dynamic due to multiple factors such as user behaviours, unpredictable changes like lighting/noise due to environment, diverse data measurement setup, etc., hence often requiring model adaptation to avoid performance degradation [168]. However, this paves the way to more practically and securely incorporate smart and intelligent devices in our lives. This is also reflected in the projected market shares of edge AI chips, where they are expected to be a 51.6 billion dollar industry by 2025, 3.5× compared to cloud AI chips [54]. Data-efficient deep learning at the edge alongside efficient HW platform design will be crucial to enable the feasibility of futuristic applications such as Metaverse [181], Digital twins [235], Fully autonomous driving [32], Novel view synthesis [212], etc. In this thesis, we mainly focus on tasks at the edge performed through DL in a data-efficient and hardware-efficient manner to achieve good overall performance.

1.1.1 Data-efficient deep learning

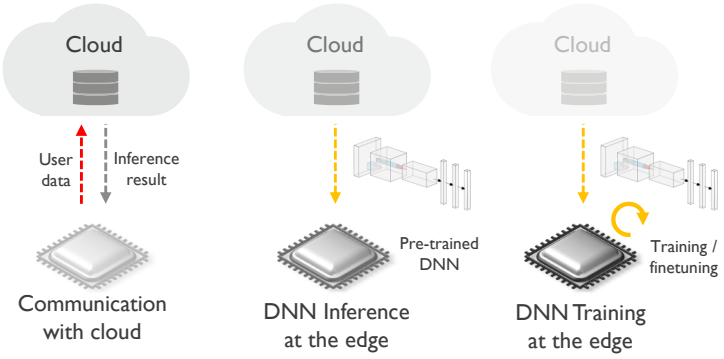
With deep learning at the edge, data-efficiency is an important metric for training Deep Neural Networks (DNNs) [111]. While there is abundant raw data available at the edge through many IoT devices, we require labelled data to perform supervised DNN training. Hence, the training samples need to be manually annotated to personalize the DNN model for the desired task. This



(a) Jumping activity sensor data (2 different users–P1 and P2) [70].



(b) Keyword "yes" audio waves (2 different users–P1 and P2) [70].



(c) Different DNN training/inference deployment scenarios. Adapted from [79].

Figure 1.3: (a) and (b) Data variation at the edge (2 users with 2 separate attempts). Used directly from [70]. (c) Deep learning deployment configurations [79].

process can often be laborious, expensive and/or not feasible [228]. However, this intermediate step of including training samples at the edge could be critical, without which the performance may suffer due to data drift and source/target data distribution shifts [164]. Figures 1.3a and 1.3b illustrate the variation of mobile sensor data based on different users/devices/attempts – multiple training samples/readings for the same user are similar, while when compared to different users are quite different. This further motivates why we need to adapt the models with user-specific data for personalization [70].

Figure 1.3c depicts the commonly used edge-cloud collaborative DL deployments [79]:

1. **Cloud-based inference:** This first configuration involves sending the data to the cloud to run inference and then only get back the result. However, it may not be feasible to offload the training process of user data to the cloud due to several reasons such as privacy, latency, connectivity, etc.[30, 79].

2. **On-device inference:** In this configuration, the trained model is downloaded to the edge device, which can then run efficient inferences locally without any communication to the cloud. However, there is no possibility to adapt the model.

3. **On-device inference and training:** This is the recent trend for DNN deployments, where the local device can in addition to performing inference also adapt its model as desired for better performance. The adaptation can be from a randomly initialized model (DNN training) or after getting a pre-trained model (DNN fine-tuning).

There exists many data-efficient DL training algorithms which might be suitable based on different factors [63, 164, 165, 228]: How different are the source and target data distributions? How many training samples do we have? Do we have new tasks / classification labels that we have never seen? We will explore more about the algorithms later in chapters 4 and 5. A common deployment configuration that is used is to utilize pre-trained models as an initialization, trained using non-personal large corpus of data, and then perform adaptation with user data on-device. However, the computational requirements at the edge increase tremendously when we move towards performing both inference and training on-device. This reiterates the need for performing the DNN computations in a hardware-efficient manner. A combination of these edge device capabilities for local learning, along with efficient learning algorithms can yield a very smart device adapting to the needs of the user.

1.1.2 Hardware-efficient deep learning

Large efforts have gone into performing efficient DL on HW, especially with the recent scaling of model and compute requirements [11, 210]. This becomes more critical for edge devices where the resources are even more constrained, demanding effective HW-efficient methods to achieve good performance. This has been investigated across the whole stack of circuit, architecture and algorithms, to satisfy target specifications [210]. Algorithmic research has explored methods such as quantization or pruning [105] (Reduced-precision computations), Neural Architecture Search (NAS) [184] (Efficient HW-aware models), Depth-wise Convolutions [34] (Parameter-optimized operators) and several other techniques [11].

These explorations also vary depending on the target HW processors such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Instruction-set Processor (ASIPs) / Application-Specific Integrated Circuits (ASICs), etc.

These typically exhibit multiple trade-offs across metrics such as flexibility, energy-efficiency and performance [104]. Programmability and design time/ease are also important factors to be considered for an optimal choice [24]. For accelerating AI workloads which predominantly involve embarrassingly-parallel tasks, domain-specific architectures are being designed with these key optimization strategies: increase performance through parallelism and improve efficiency through specialization [44, 177, 242]. However, a balance between the two is important, as being too specialized will lead to its obsolescence in a domain of continuously evolving algorithms [156]. Multiple accelerators can also lead to a decrease of system-level efficiency due to communication overheads between different specialized cores [8]. Efficient architectures are also being implemented through custom instructions for general-purpose HW to target AI workloads more efficiently such as: integer matrix-multiply and accumulate (IMMA) instructions in NVIDIA V100 GPUs [35]; integer dot-product or matrix-multiply and accumulate vector instructions in ARM SVE-based processors [204], etc. The choice of HW often depends on multiple requirements, while typically the trend is moving towards heterogeneous systems and multi-core designs with different cores suitable for different workloads, with the aim of efficient overall computation. However, this does increase the complexity of efficiently mapping workloads and enabling high utilization, which are challenging tasks still being explored [219].

In conclusion, multiple variants of HW-efficient deep learning are possible with not yet any single universal solution that works best for all scenarios. However, the most benefits are obtained when algorithms and HW are co-designed together. Additionally, maintaining an optimal balance of specialization and generality in an instruction-programmable manner will lead to efficient computations while continuing to be relevant in a quickly evolving deep learning algorithm space.

1.2 Open research questions

While hardware-efficient and data-efficient deep learning at the edge is very beneficial, leading to the enablement of many applications once thought infeasible, there are many challenges that still need to be solved. We explore the following research questions in this thesis:

Q1. How to efficiently deploy data-efficient deep learning algorithms on resource-constrained hardware?

Efficient DNN training at the edge has been an attractive feature to have and has hence been researched extensively across diverse approaches. As previously discussed backpropagation is the backbone for DNN training algorithms, however, it is a highly compute and memory intensive process. There have been papers trying to overcome the overheads of backpropagation algorithms to make them more efficient [62, 86, 149, 154], but an alternative algorithm with equivalent performance and generalizability for training tasks has not yet been developed. One of the prominent techniques for making backpropagation-based DNN training efficient has been through quantization or low-precision training [46, 51, 60, 207, 208, 231]. Different dataformats have been explored to efficiently compute DNN operations on hardware: Floating point (FP), Fixed-point (FXP), Logarithmic, Posit, etc. [3], to obtain a better trade-off of algorithmic accuracy and hardware efficiency. A specific format which was popular for accelerating DSP systems, called block floating point (BFP) [161] is now gaining traction for DNN deployments as part of a larger umbrella of formats called as Microscaling (MX) formats [47]. These are defined as a hybrid of FP and FXP datatypes, wherein a group of FP/FXP numbers share a common power of 2 exponent, which is very hardware-friendly for scale handling.

In this thesis, we explore such MX or block-scaled quantization for diverse but data-parallel DL application scenarios at the edge. To accommodate different requirements of accuracy-performance trade-offs, we investigate the impacts based on different block sizes, shapes and precisions. Depending on the task modality, the literature approaches for data-efficient algorithms varies, and are therefore discussed in separate sections as follows:

- **Data-efficient image classification tasks:**

Image classification is one of the major tasks at which AI or DL outperformed humans [109]. However, the traditional DL algorithms struggle to perform well when the training data is very limited [58, 228]. This is especially true compared to humans who have generalized learning abilities for any unseen task even with limited samples. For example, consider a classification task of images of any 5 numbers (class labels). Even if there are only 1 or 2 labelled training samples available, we would have a high accuracy of identifying the correct test number, while the DNN model suffers. To enable good DNN performance on new tasks with less data, several few-shot learning (FSL) algorithms have been explored [63, 228]. With techniques like optimizer-based meta-learning, we try to learn a suitable initialization or other optimizer parameters that would enable good training convergence on any new few-shot task [58, 152]. This

method has also shown promising results for tasks like mobile sensing, related to personalization for activity and speech recognition with diverse users and mobile devices [70]. Additionally, these algorithms are also model-agnostic, making them applicable to diverse learning algorithms. Since this also conforms to the "pre-train – initialize – fine-tune" mechanism for on-device learning, we investigate optimizer-based meta-learning for FSL applications. For scenarios where we have more training data and still needing model retraining to adapt to a new target domain / task, transfer learning algorithms can be used effectively with a similar strategy of fine-tuning on-device [164]. These are different from conventional large-scale DNN training at the cloud where there is no pre-training and the precision constraints for accurately representing the parameter distributions throughout all training iterations can be more demanding [61]. Hence, the precision constraints explicitly for fine-tuning small models are evaluated in this thesis chapters 4 and 7 through block-scaled quantizations.

- **Sample-efficient continuous control tasks:**

Continuous control tasks of systems like robotics require different type of algorithms than conventional supervised learning algorithms, in that they need to learn from interactions with an environment. Reinforcement Learning (RL) [209] algorithms have been used for such tasks where sequential decision-making is needed to achieve a desirable outcome. DL and RL have been used together for Deep Reinforcement Learning showing tremendous potential recently across multiple applications [6]. There are mainly 2 variants of such RL – Model-free RL (MFRL) and Model-based RL (MBRL). It is now typically shown that MBRL algorithms which use a predictive model of the world (through interactions), are orders of magnitude more sample-efficient than its counterpart MFRL [38] which learn a control policy without any model, and hence would be our main focus. The study of HW-efficient explorations through quantization is much more scarce in the RL domain [15, 74, 107]. These workloads typically involve parameters in continuous spaces and the models themselves are usually regression models, making them less resilient for quantization when compared to tasks like image classification tasks. Additionally, MBRL models typically use DNN-based dynamics models to predict the state transitions in a system. They commonly exhibit the compounding error problem when used for planning and control with long horizon predictions [95]. We further explore the impacts of DNN quantization and compounding error on the control quality, with the aim of moving towards HW-efficient planning in MBRL in chapter 5 of this thesis.

Q2. How do we efficiently deploy DNNs with block-scaled formats using general-purpose vector instructions?

Datatypes such as block-scaled formats are becoming very popular for low-precision DNN deployments demonstrating tight HW-SW co-optimizations for both inference and training, to an extent that there are propositions to standardize them as a low-precision data-format for AI [160]. However, they are still predominantly explored through custom accelerator deployments [43, 45]. This prompts the question of whether we can still utilize these formats on general-purpose devices that are more widely available and are easily programmable. Several different vector/SIMD instruction sets (ISAs) are available: Intel AVX-512 [41], ARM NEON [131], AMD XOP [1], etc., that are used across server and embedded devices with varied vector/SIMD lengths. Another advantage of general-purpose ISAs is the availability of development support through multiple libraries for DNN deployment – many for inference and few for training, but typically only for standard datatypes such as FP32, FP16 and INT8 [123, 125, 142, 159, 233]. For non-standard DNN operations, high-performance math libraries and kernel implementations would also be generally available. There also exist some ISAs that are vector-length agnostic (VLA) such as ARM SVE [204] and RISC-V Vector (RVV) [59], which enable portability across varied HW vector lengths. This greatly reduces the SW development efforts and enables quick design space explorations. These benefits also further justify manual implementations in assembly or intrinsics for designing high performance code, in scenarios where compiler auto-vectorization was not very successful. Exploring custom kernels for block-scaled GeMMs might enable more computationally and memory efficient DNN deployments on general-purpose processors, by reducing FP computations, while still maintaining good performance. However, the exact benefits depend on the implementations to overcome the overheads of not using custom instructions. But, this would also provide insights on instruction-set extensions to move towards more efficient block-scaled computations. Chapter 6 of this thesis investigates the implementations of block-scaled INT8 GeMMs using ARM SVE instructions with a VLA programming model.

Q3. How do we increase efficiency for block-scaled DNN deployments by moving towards domain-specific instruction set processors?

As discussed in the previous question, there exists multiple block-scaled custom accelerator designs, a majority of which are only design-time configurable (ASIC-like) and have limited deployment flexibilities. The other extreme design configuration is to implement flexible block-scaled datatypes using only general-purpose vector instructions, while still utilizing sub-word level parallelism across

variable vector lengths (CPU/DSP-like). Since we only consider general-purpose instructions, this is more like an emulation to implement the same functionality of block-scaled datatypes, similar to hard-float vs soft-float implementations for FP operations. However, not all block-scaled or MX formats are suitable for emulation due to the lack of corresponding MAC instruction support, which might be available/made as stand-alone HW units [191]. An efficient and impactful implementation would hence require instruction extensions to further enhance deployment possibilities at some slight HW overhead. Therefore, there is a clear trade-off of flexibility vs efficiency with high potential for HW-SW co-optimizations enhanced by open-source vector processor architecture such as Ara / Spatz [25, 171]. We[1] investigate more tightly co-designed domain-specific instruction-set processors (DSIPs) for data-parallel DNN workloads with block-scaled datatypes in Chapter 8 of this thesis.

1.3 Thesis organization and contributions

A primer on the concepts used in this thesis is provided in chapters 2 and 3, to enable a better understanding of the rest of the thesis. **Chapter 2** details on the functionalities of deep neural networks (DNNs), its different stages – inference and training – to enable effective performance, and our target data-efficient application scenarios at the edge. **Chapter 3** delves into common techniques to enable efficient HW-SW co-optimized DNNs. It also explores various quantization strategies for DNN acceleration, with more focus on block-scaled or MX datatypes.

Figure 1.4 depicts a high-level overview of the scope of this thesis, along with annotations of our chapter-wise contributions.

Chapter 4 addresses the challenges of data-efficient learning on a previously unseen image-classification task on resource-constrained hardware. We propose a modified pre-training approach called "Learn to learn on chip". It combines the 2 techniques of meta-learning (Learn to learn) and quantized training (Learn on Chip). Through our approach involving an upfront inclusion of target hardware constraints during the meta-learning process, we obtain a better or more suitable initialization for few-shot learning with limited resource constraints on the target hardware. This chapter is based on the following publication:

1. Nitish Satya Murthy, Peter Vrancx, Nathan Laubeuf, Peter Debacker, Francky Catthoor, and Marian Verhelst. *"Learn to Learn on Chip: Hardware-aware Meta-learning for Quantized Few-shot Learning at the*

[1] In collaboration with EPFL

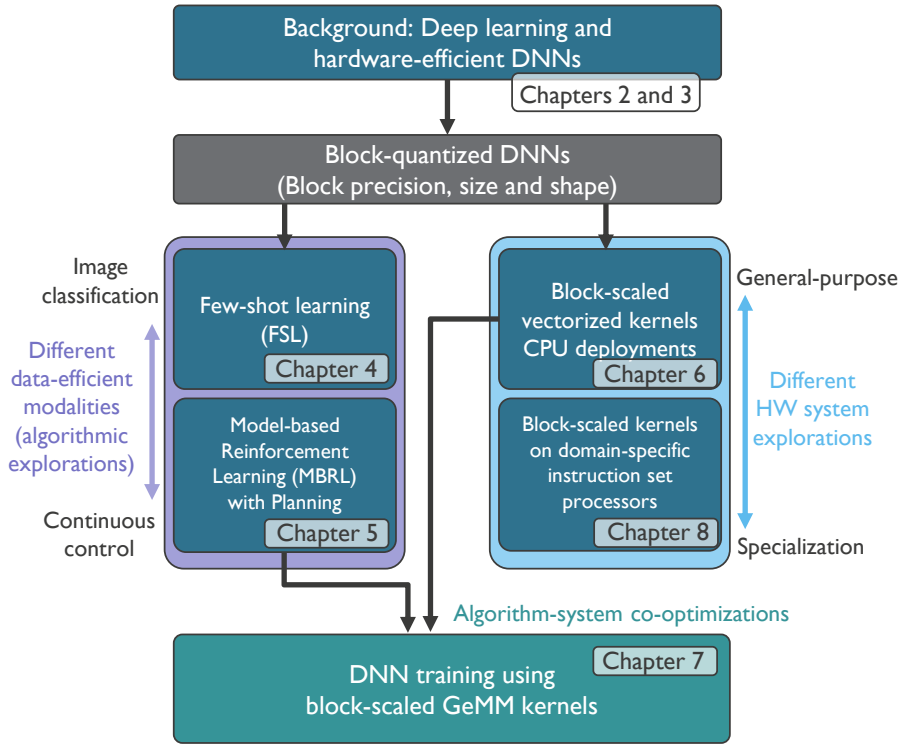


Figure 1.4: Overview of thesis contributions.

Edge." In 2022 IEEE/ACM 7th Symposium on Edge Computing (SEC), pp. 14-25. IEEE, 2022 [141].

Chapter 5 explores the continuous control task applications through data-efficient RL algorithms like model-based reinforcement learning (MBRL).

We look into the impacts on the planning process, using pre-trained learned dynamics model, with quantization through block-scaled inferences. It is important to note that this is conceptually different compared to conventional DNN inferences which are predominantly single-shot. In contrast, planning requires sequential DNN inferences over time, wherein quantization error can compound leading to potentially larger degradation, and hence require more attention during quantization. This chapter is based on the following published paper:

1. Nitish Satya Murthy, Francky Catthoor, Marian Verhelst and Peter Vrancx. "Quantized dynamics models for hardware-efficient control and planning in model-based RL." In: Machine Learning and Principles and Practice of Knowledge Discovery in Databases. Ed. by R. Meo and F. Silvestri. Cham:Springer Nature Switzerland, 2025, pp. 196–209. [195].

Chapter 6 investigates the design and optimization of block-scaled GeMM kernels using general-purpose vector instructions without customization. We define a methodology to deploy such GeMMs using available ARM SVE instructions in an efficient manner with good utilization through i) Vectorized microkernel intrinsics. ii) Flexible block sizes and shapes (1D and 2D). iii) Dynamic and static requantization with fused kernels. We also highlight the comparison with standard datatype deployments such as INT8 and FP32 to highlight our kernel's performance. This chapter is based on the following publication:

1. Nitish Satya Murthy, Francky Catthoor, and Marian Verhelst. "Optimization of block-scaled integer GeMMs for efficient DNN deployment on scalable in-order vector processors." Journal of Systems Architecture 154 (2024): 103236 [194].

Chapter 7 analyzes the impact of utilizing block-scaled kernels for DNN training of Fully-connected (FC) network architectures. Similar networks are analyzed for deployments that were earlier evaluated for data-efficient scenarios. We additionally demonstrate the impacts due to flexible block shapes, which are more pertinent for training scenarios, and showcase the resulting optimizations that can be performed. Lastly, we introduce a methodology that involves combining the two contrasting training configurations – coarse blocks but HW-efficient training and fine blocks but HW-inefficient training, to obtain an overall HW-SW co-optimized deployment. This chapter is based on the following publication:

1. Nitish Satya Murthy, Nathan Laubeuf, Debjyoti Bhattacharjee, Francky Catthoor, and Marian Verhelst. "Adaptive block-scaled GeMMs on vector processors for DNN training at the edge". In: 2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC) [140].

Chapter 8 moves towards explorations on domain-specific instruction-set processors (DSIP) capable of handling block-scaled datatypes with the aim of efficient HW-SW co-optimization. In the first subsection, we explore SoftSIMD

type processors, which are highly flexible SIMD microarchitectures that are suitable for block-scaled DNNs to reach specific trade-off points. We explore different mixed-precision configurations, block choices, their mapping and deployment metrics for standard CNN tasks.

Lastly, in **Chapter 9** we conclude the thesis with a summary of our contributions and also highlight the potential future explorations of this work.

Chapter 2

Deep neural network

As the name suggests, Neural Networks (NN) involve "artificial neuron" inspired by the "biological neuron" in our brains which is one of the fundamental units responsible for human intelligence [137]. The biological neurons are connected to each other through dendrites, which help navigate signals through a complex network comprised of around 86 billion neurons [84]. An Artificial NN is designed to have a similar network of artificial neurons, where each individual neuron (called perceptron) outputs a non-linear activation function of the weighted sum of its inputs, to the next neuron. Multiple layers of such perceptrons are connected to form a multi-layer perceptron (MLP) or fully-connected (FC) NN as illustrated in Fig. 2.1. Apart from the input and output layer, the intermediary layers are also called hidden layers. When the number of hidden layers is lower than 2, the networks are often called shallow NNs while the alternative is referred to by deep NN (DNN). In this manner, a complex non-linear mathematical function can be performed, with the capacity to model a variety of problems. It is important to note that the linear activation functions with any number of layers can all be reduced to a single linear operation, which does not help in varied learning capacity. Hence, the non-linear activation part is crucial for model learning.

These DNNs are the major reason for the growing popularity in AI. A basic DNN involves tunable parameters – weights and biases, in a network architecture of interconnected neurons. However, irrespective of the DNN architecture, a common methodology is followed to perform supervised training of DNN that enables it to do certain target tasks (classification, regression, etc.).

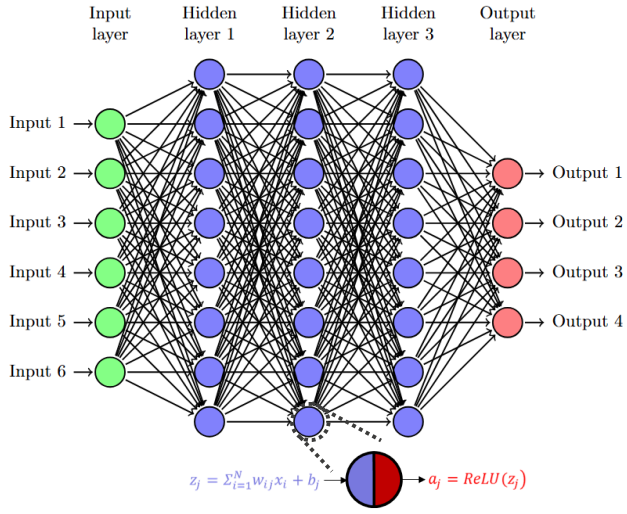


Figure 2.1: Structure of a NN / MLP with 6 inputs, 4 outputs and 3 hidden layers (8 neurons each). Activation function depicted here is a ReLU function. The functionality of an individual neuron or perceptron to perform a non-linear activation on a weighted sum of its inputs is also highlighted.

2.1 DNN inference and training

The general methodology for training a DNN can be described in 4 different steps. These steps 2.1.1, 2.1.2, 2.1.3 and 2.1.4 are usually done in a sequential loop for a particular mini-batch (B) of the total available training data, which is called a training iteration. When the complete training data is passed to the DNN across multiple batches, a training epoch is completed. Typical DNN training requires 100s of epochs for good convergence, while fine-tuning of DNNs from pre-trained model initializations might require only 2-5 epochs.

2.1.1 Forward propagation

Forward propagation mainly involves computing the NN outputs for the given inputs. The network is initialized with random weights, based on specific distributions as prior research has indicated strong impacts on training convergence due to initializations [117]. For the 3-layered network illustrated in Fig. 2.1, the following equations describe the vectorized computation of a feed-forward propagation with a batch size - B training samples:

$$A^{[1]} = X \tag{2.1}$$

$$Z^{[2]} = A^{[1]} \cdot W^{[1]} + b^{[1]}; \quad A^{[2]} = ReLU(Z^{[2]}) \tag{2.2}$$

$$\vdots$$

$$Z^{[5]} = A^{[4]} \cdot W^{[4]} + b^{[4]}; \quad O = A^{[5]} = Softmax(Z^{[5]}) \tag{2.3}$$

where:

X – Input $(B \times 6)$
$W^{[m]}, b^{[m]}$ – Weight matrix $(6/8 \times 8)$ and Bias vector (1×8) of the mth layer
$Z^{[m]}, A^{[m]}$ – Weighted sum $(B \times 8)$ and Activations $(B \times 8)$ of the mth layer
O – Final output $(B \times 4)$
$ReLU$ – $\max(x, 0)$

The output of the non-linear model is given by Eq. 2.3. The part of the output before a non-linear function is also referred to as logits. For a classification task, often a Softmax function is applied to convert the logits into predicted probabilities for classification into different labels. Depending on whether X is a vector (1 sample) or a matrix, the forward propagation mainly involves General Matrix-Vector or Matrix multiplication (GeMV or GeMM) operations.

Majority of the end-users typically use DNNs only in the forward propagation mode where its weight parameters are fixed and not updated, also known as the **inference** process. Essentially, this is equivalent to performing a complex mathematical function (trained DNN) on the inputs to get corresponding outputs, with additional processing depending on the desired task (classification / regression). While this example discussed a simple DNN with only MLP or FC layers, modern networks involve a variety of layers and operations, a few of which are discussed as follows.

Popular DNN layers

The most common DNN layers are **FC** and **convolution** layers which have operations as described in Fig. 2.2. These layers can be represented as nested for-loops or different multiply-accumulate (MAC) operations. Both of them can also be computed as a GeMM operation, directly for FC as we discussed previously and after *im2col* transformations for convolutions. This is a common approach for deployments since there exist a lot of standardized libraries that

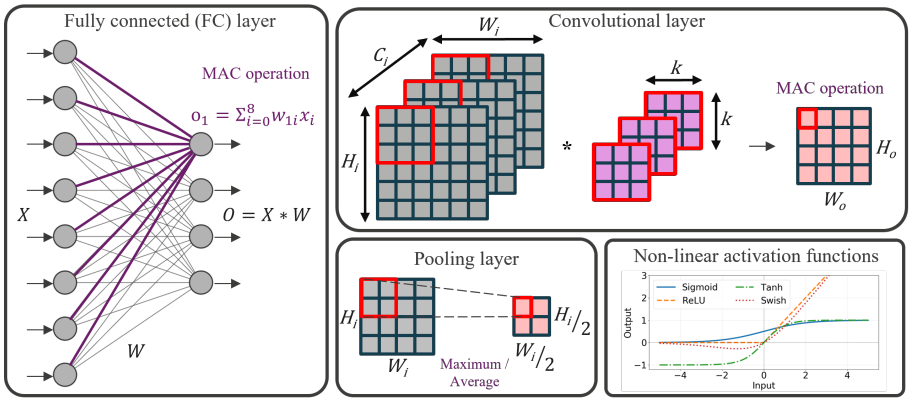


Figure 2.2: Common DNN operations.

optimize GeMM computations [217]. Convolutions typically involve "shared weights" known as *filters* as opposed to FC layers, which is the reason for performing *im2col* transformations before a GeMM. Each filter is unrolled into a vector of the weight matrix for all filters. While for activation or feature maps, different patches (same shape as a filter) are unrolled into corresponding rows of the activation matrix.

Pooling layers are typically used to downscale the feature map size for deeper layers to reduce computational overheads, while not removing too much information that degrades performance. Typical pooling layers reduce the width and height of the feature maps by averaging or propagating only the maximum or average of every 2×2 grid known as Maxpooling and Average pooling respectively [64].

Activation functions used at the neuron output, need to be a non-linear function as we previously discussed. Most popularly used activation functions are: Sigmoid [80] and ReLU [146] function. Few key justifications for these selections out of many other functions [157] are differentiability of the functions, ease of training, stability of training convergence, etc.

Batchnorm operations are intended to be used for better convergence by normalizing the activation values, making the loss surface more regular and easier for the optimization process to move towards a minima [91]. The process involves calculating the mean and variance of input mini-batch data as shown in Eq. (2.4). For multiple mini-batches, commonly observed during training, running averages of each mini-batch mean/variance are tracked with more importance for recently observed batches. With these parameters, the inputs are normalized accordingly, and further scaled and shifted using Eq. (2.5) to

finally produce the batchnorm outputs. During inference, the statistics estimates are typically frozen.

$$\text{Mean/variance:} \qquad \mu_B = \frac{1}{B}\sum_{i=1}^{B} x_i, \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_B)^2 \tag{2.4}$$

$$\text{Normalize, scale and shift:} \qquad \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\ , \qquad y_i = \gamma \hat{x}_i + \beta \tag{2.5}$$

Where x_i and y_i are inputs to and outputs of the batchnorm operation, respectively. B is the mini-batch size. ϵ is a small constant for numerical stability. γ and β are learnable model parameters for scaling and shifting.

2.1.2 Loss calculation

Since we perform supervised learning, typically each input has a target output. Hence, a loss metric ($L(\theta)$) can be calculated between the DNN model (parameters θ) outputs and the corresponding targets, for all samples in a mini-batch B. Depending on the task and objective, different loss functions may be more compatible and need to be selected by the user beforehand. Eq. (2.7) represents the popular loss functions for classification tasks (Negative log-likelihood loss - NLL) and regression tasks (Mean squared error - MSE).

$$\hat{y}_i = f_\theta(x_i), \qquad x_i : \text{input},\ f_\theta\text{: DNN with parameters } \theta,\ y_i : \text{Target} \tag{2.6}$$

$$L_{NLL}(\theta) = -\sum_{i=0}^{B} y_i \cdot \log(\hat{y}_i) \quad | \quad L_{MSE}(\theta) = \frac{1}{B}\sum_{i=0}^{B} ||y_i - \hat{y}_i||_2^2 \tag{2.7}$$

2.1.3 Backward propagation

The key mathematical foundation that enables backward propagation and consequently DL, is the chain rule of derivatives which defines the method for calculating the derivatives for a composite function such as: $\frac{dy}{dx} = \frac{dy}{dg} \cdot \frac{dg}{dx}$. Since DNNs mainly involve multiple composite functions in a large network architecture, the relation of the calculated output loss to each individual tunable DNN parameter can be derived through the aforementioned chain rule.

As the name suggests, backward propagation or backpropagation involves computations moving in the backward direction, where we get inputs from deeper layers starting with the final loss and compute outputs towards the shallow layers. For the chain rule to be used effectively, all operations in the forward pass need to be differentiable (or at least approximated). The easily scalable model capacity of DNNs combined with an automated learning algorithm like backpropagation has led to tremendous performances across multiple domains. The equations for backpropagation for FC layer types are as follows at a particular layer L:

$$A^{[L+1]} = A^{[L]} \times W^{[L]} \quad \rightarrow \quad \text{Forward prop / Inference} \tag{2.8}$$

$$G^{[L]} = (A^{[L]})^T \times E^{[L+1]} \quad \rightarrow \quad \text{Weight gradient calculation} \tag{2.9}$$

$$E^{[L]} = E^{[L+1]} \times (W^{[L]})^T \quad \rightarrow \quad \text{Error gradient calculation} \tag{2.10}$$

As seen in Eqs. (2.9) and (2.10), the backward operations for FC layers still mainly involve GeMMs but with transposed operands. The activation function and biases were omitted to simplify the equations. The process is similar for convolutions: to calculate weight gradient we need to perform dilated convolutions of reorganized input tensor with output error gradients; and we need to perform transposed convolution on the output error gradient by the convolving kernel to obtain input error gradient [237]. While these analytical equations exist for calculating gradients for some key DNN operations, there are standard frameworks such as PyTorch [166] and Tensorflow [135] among others, that automatically calculate all gradients by traversing backward on a graph that is built for every learnable parameter based on all of its forward operations.

2.1.4 Weight update

With gradients of every parameter available now, we know the direction in which the high-dimensional weight parameters need to change to move towards a minima in the loss landscape. The magnitude of the steps taken during the weight update is a critical parameter which greatly influences the training convergence quality. Large steps can lead to divergence with increasing losses, while small steps can be the cause of extremely slow convergence. Hence, an optimal step size, defined by the term "learning rate", needs to be tuned empirically. Based on this the weights can be updated as shown in Eq. (2.11), which is called the Stochastic Gradient Descent (SGD) step.

$$W^{[L]} \leftarrow W^{[L]} - \eta \cdot G^{[L]}, \quad \eta \rightarrow \text{learning rate}, \quad G^{[L]} = \frac{\partial L}{\partial W^{[L]}} \tag{2.11}$$

Additional issues can lead to difficulties in training convergence such as getting stuck at local minima, saddle points, plateaus, etc. [206]. To resolve these issues, multiple DNN optimization algorithms have been developed, with some of the most popular ones being: SGD [101], SGD with momentum (2x memory overhead than SGD) [128], Adam (3x memory overhead than SGD) [103], etc. Recent methods also utilize 8-bit block quantization for the optimizer parameters, that can be effectively used as a drop-in replacement to FP32 optimizers saving memory especially for large models [49]. Other optimizer modifications also involve scaling the loss functions to enable efficient low-precision training [125, 138].

Other than the previously mentioned steps for training, the concepts discussed here are also important for several scenarios:

- *Weight initializations* before the training process have been previously studied to have a large impact on training convergence qualities due to exploding / vanishing gradients based on several parameters such as layer operation type and size, activation function, etc. [83]. Several standard distribution initializations have been explored such as Uniform, Xavier [67], Kaiming [83], etc.
- *Hyper-parameter tuning* might need to be performed either manually or automatically to search suitable training hyper-parameters for better convergence [243]. The available datasets are typically split into train, validation and test splits, to enable effectively tuned network configurations and test them.
- *Regularization* is an important process which is often needed to avoid model overfitting on training dataset and not being able to generalize well on test data. This is typically done through weight decay, which penalizes the weights from being too small or large, thereby restricting the model's capacity and prevent overfitting [85].

2.2 Common DNN architectures

A DNN architecture involves the definition of multiple components. Firstly, the definition of activation function is critical for the functionality of DNNs, as these

are the main functions that provide the non-linear features. While ideally it is desirable to have a differentiable function to still conform with backpropagation, functions like sigmoid have other drawbacks such as vanishing gradients that made it difficult to train deeper networks. In these cases, Rectified Linear Units (ReLUs) were used successfully, which are not only computationally simpler but also enabled better convergence [68].

DNN architecture also involves the specification of how different neurons are interconnected or which operations to use. The universal approximation theorem suggests that a single hidden layer of NN, with no limits on size, is capable of closely approximating any continuous mathematical function within some bounds [42]. However, for practical applicability we want to learn as efficiently as possible. Additionally, depending on target tasks, some DNN architectures will be more suitable than others. The exact architecture of DNNs is up to the designer, however, in general the model capacity for learning increases for larger DNNs and the deeper models can model more complex or higher order functions. With this in mind, several different architectures have been researched, with different network connections, some of which are empirically designed while others are generated automatically based on an optimization objective through Neural Architecture Search (NAS) [53]. The rest of this section focuses on some key DNN architecture innovations that have been developed to achieve more efficient DNN performances on image classification tasks.

The earliest and simplest NNs were of **FC or MLP** type. However, they are not the most computationally-efficient for image classification tasks if we map each pixel to individual neurons. Shared weights, called filters, are found to be more effective for image workloads through **convolution operators**, reducing the number of parameters immensely, which led to Convolutional Neural Networks (CNNs) [116]. **AlexNet**, a CNN-based architecture, was one of the first DNNs to achieve high accuracy on a complex image classification task without any manual pre-processing, further revolutionizing the idea of DL [109].

Even with CNNs, different architectures have been explored, but there was a difficulty in learning effective representations with increasing network depths, even though the theoretical model capacity was supposed to be better. This was due to vanishing gradients, as the magnitude of gradients reduced to very small values during backpropagation to shallow layers. To resolve this **ResNets** [82] were invented wherein the residuals across blocks were used to provide separate paths for gradients to flow avoiding this vanishing gradient issue. This enabled efficient CNN training for even 100/150-layered very deep networks, with performances even outperforming humans.

Additional optimizations based on parameter count and MACs was done by using depth-wise separable convolution in **Xception** [34] instead of conventional

convolutions. This involves first performing a depthwise convolution and then a pointwise convolution. Compared to standard convolutions, the depth-wise separable convolution convolutions have reduced parameter counts and MAC operations, while maintaining exact dimensions of outputs. These layers also became popular for mobile deployments through **MobileNets** [88] due to their lightweight and efficient models.

Apart from these manual or empirical architectural improvements, NAS has also been used to generate DNN architectures such as **ProxylessNAS** [23] and **MCUNet** [124] that automatically search and generate network architectures, based on target optimization objectives with certain user constraints.

While we have mainly discussed MLPs and CNNs until now, there are other popular architectures as well which had their own evolution for specific tasks such as time-series data. RNN [188], LSTM [72], architectures were designed for handling sequential inputs by learning long-term dependencies through feedback loops. However, these suffered from being able to capture long sequences due to the serial nature of training limiting scalability. This was improved by a concept called attention, which enabled understanding relations among different positions in a sequence, creating an architecture known as Transformers [218]. Transformers have now emerged as a very popular architecture capable of performing a wide variety of tasks, and being used predominantly for foundation models [250]. These models are typically much larger than the previously discussed MLPs and CNNs, and are now also being researched on techniques to deploy them on edge devices effectively [227]. In the scope of this thesis, we mainly focus on MLP and CNN architectures.

2.3 Conclusion

In this chapter, we briefly looked at the origin of NNs, introduced the definition of DNNs forming the domain of DL, which is a significant reason for the proliferation of AI.

We also explained some of the commonly used layers and operations in modern DNN architectures. The process of training a DNN to perform a task based on *supervised learning* with known targets was extensively discussed. The evolution of DNN architectures leading to continual improvements of DL performance was walked through.

Note that this chapter mainly explored *supervised learning* concepts most relevant to the thesis, while other learning techniques are also being actively researched. *Unsupervised learning* aims to learn from data without needing any

target labels through pattern identification in the data, with popular methods such as clustering [200] and dimensionality reduction [216]. *Reinforcement learning* algorithms are used in decision-making scenarios, by rewarding desirable actions and penalizing the others to eventually learn a policy for autonomous control [209]. Another domain called *neuromorphic computing* deals with exploring algorithms that move towards the biological learning category, speculated to take place in human brains [187]. In this thesis, we mainly explore *supervised* and *reinforcement learning* algorithms, particularly in the context of DL at the edge. The next chapter focuses on more HW-efficient DL algorithms that enables DNN computations even on resource-constrained devices.

Chapter 3

Hardware-efficient DL

The previous chapter introduced DNNs and its training methodology, while in this chapter, we delve deeper into techniques for deploying DNNs efficiently. We saw previously in Chapter 1 regarding the increasing computational requirements due to DNN scaling demands, and the motivations to deploy them at the edge where resources are constrained. Both of the previous points demand an exploration of hardware-efficient DNN techniques. To achieve this, multiple algorithm and HW platforms have been researched, some of which are briefly discussed here.

3.1 Efficient DNN algorithms

A substantial body of research has focused on developing efficient DNN algorithms. An overview and taxonomy of recent advancements are presented in [11], which enables easier understanding and identifying opportunities for further improvements. They categorize different methods based on fundamental questions such as where, what, when and how specific optimizations are performed, as described in Fig. 3.1. Based on this, we highlight some of the most widely used algorithms:

- **Non-backpropagation techniques** (Replace training algorithm)

While DNN itself was inspired by a biological neural network, the learning algorithm in the brain has largely been argued to be different from backpropagation [122]. Although backpropagation is a simple yet powerful technique, it has major limitations that make them biologically implausible [62]:

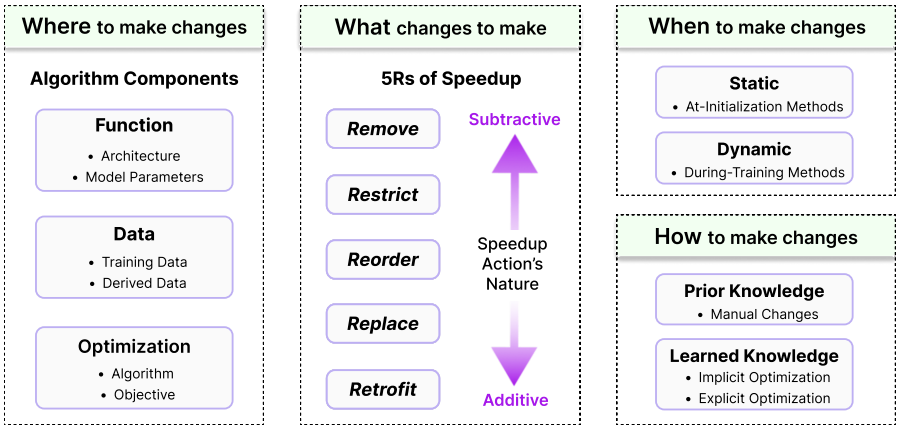


Figure 3.1: Taxonomy of efficient DNN algorithms. Taken from [11].

Weight-transport relates to needing to use the same weights in both the forward and backward (transposed) pass; Update-locking refers to the challenge of updating a layer's weights only after the complete forward pass and the backward pass until that layer is performed. These two major issues also led to making backpropagation a very compute and memory intensive process. FA [121], DFA [154], and DRTP [62] overcome the above challenges by using random fixed matrices during backpropagation, and enabling weight updates by directly bypassing the complete network propagation. While these techniques have reduced some of the complexity associated with backpropagation, their performance and scalability have yet to reach the desired levels. This remains an active area of research, with ongoing efforts to develop highly efficient algorithms that match the learning capabilities and energy efficiency of the brain [86, 134].

- **Curriculum learning** (Reorder training data)

Curriculum learning deals with changing the ordering of training data in specific ways, deviating from conventional random ordering, to accelerate the overall training process. This is inspired by human education process, where we first learn the easier tasks in earlier years and gradually move towards more difficult versions. Several works have reported on benefits of this, a survey of which is presented in [226]. This could be an interesting option, especially with the recent highlights of the importance of data-centric solutions [150].

- **Neural architecture search (NAS)** (Replace DNN architecture)

We briefly discussed in the previous chapter regarding the importance of DNN architectures to more efficiently perform a specific task. For efficient

deployments on resource-constrained HW specifically, architecture design is even more crucial. The general trend in increasing model sizes and parameter counts results in increased learning capacities, but at the cost of more demanding deployments. The performance improvement is however not ever-increasing, rather encountering saturation after some model sizes. NAS aims at finding optimal architectures that meet specific requirements automatically without much human intervention.

NAS typically involves 3 components: A search space, a search strategy and a performance estimator [53]. Prior knowledge about architectures could be inputted to the search space possibilities to limit the available options and speed up the process. Several search strategies such as RL, evolutionary algorithms [248], bayesian optimization [229], could be used to find more optimal configurations within the search space. Performance estimator forms the optimization objective, for which we aim to find optimized DNN architectures. These performance measures could be a simple accuracy metric on some datasets; multi-objective requirement with trade-offs of accuracy, latency, and energy-efficiency [37]; HW-aware for effective HW-SW co-designs [12]. Since the complete optimization procedure can be very time-consuming, techniques such as one-shot NAS are gaining popularity [21].

- **Normalization techniques** (Retrofit DNN architecture)

DNN architectures can also be improved from making the optimization process better. Normalizing the inputs of a training batch through batch normalization has been shown to accelerate the training process by up to 14 times [91]. Normalizing ensures a more regular loss landscape and enables the optimizer to perform weight updates more effectively with larger learning rates. This is often performed by tracking statistics such as updating the running mean and variances of the batch activation values throughout the training process. Batch normalization layers are ubiquitous in modern DNN architectures, with also several variants being proposed such as Instance Normalization, Group Normalization, etc., [232] for task-specific improvements and Range Normalization [9] for more HW-efficient deployments. Lastly, since inference process does not involve any optimization, it is common to remove these layers by a batchnorm folding process [92] to reduce computational complexity.

- **Pruning** (Remove model parameters)

Pruning is a frequently-used algorithm for DNN compression, which is performed by removing parts of the network that are less critical while not incurring much accuracy degradation. It is also used to induce sparsity in a network especially for HW platforms that can capitalize them effectively. Papers have demonstrated maintaining similar accuracies with up to 60-80% of network pruning, however, the exact benefits are dependent on the particular architecture, pruning method

and the specific target task [176]. Pruning can also be applied statically after training [81], dynamically during training [133], in an Energy-aware manner [238], etc. Pruning can be applied at different granularities (neurons, filters, channels, etc.), or in a structured / unstructured approach, which are covered extensively in this survey [120].

- **DNN quantization** (Restrict model parameters)

Quantization is one of the most popular forms of achieving HW-efficient DNNs. The success of quantized DNNs can be attributed to the fact that models can achieve good algorithmic performance even with low precision, including extreme scenarios such as binary quantized models [174]. Quantization can be used during both the inference and training process, and can be applied to different parameters – model parameters (weights), intermediate activations, etc. Quantization process also ties closely to the HW platform used for acceleration which is therefore used extensively for co-design approaches. This topic is explored in-depth in the following section 3.2, as this is predominantly used in the thesis.

3.2 Quantization

Quantization is a general mathematical term, also outside the DNN context, defined as the process of reducing the number of permissible values from a large set to a smaller set. Following the brief motivation for quantized DNNs in the previous section, it is one of the most popular techniques for HW-efficient deployments. Quantized DNNs are also referred to as low-precision DNNs, essentially due to the reduced precision effect. Any digital number has limited number representation possibilities as they are described by discrete binary strings.

3.2.1 Different datatypes for DNNs

A specific number system defines how each binary string is used to approximate a real number. Several datatypes are used in computer arithmetic, the most common ones being **Floating point** (FP) and **Fixed-point** (FXP) numbers. 32-bit floating point (FP32) numbers are one of the most popular standard datatype. A binary string of FP32 formats are converted to a decimal number equivalent by the Eq. (3.1) according to IEEE-754 standards [96], showing also an example of the conversion. Note that the conversion equations for special conditions such as for subnormal, zero and infinity, are ignored here for simplicity. From the equations, it can be observed that FP32 formats can

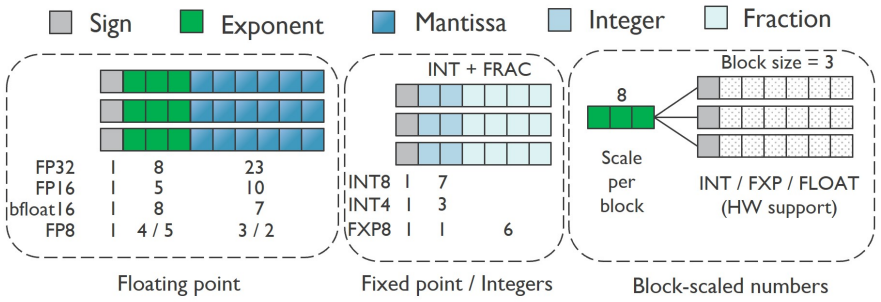


Figure 3.2: DNN datatypes - conventional and block-scaled formats.

represent very small and large numbers well. However, it has previously been studied that such high precision is often not required for every DNN parameter to maintain its algorithmic accuracy. Hence, researchers have also explored other variants and formats, as depicted in Fig. 3.2.

$$x_{fp32} = (-1)^{s}\ 2^{exp-127}\ 1.mant \qquad (s, exp, mant - \text{binary string}) \tag{3.1}$$

$$x_{binary} = 0\ 10000100\ 00011101011000100100111 \tag{3.2}$$

$$x_{fp32} = (-1)^{0}\ 2^{132-127}\ (1 + 962855/2^{23}) = 2^{5} * 1.11478126049041748 \tag{3.3}$$

$$= 35.673000335693359375 \tag{3.4}$$

Fixed-point numbers are calculated similar to the mantissa parts in FP32, however without a specific exponent part. Instead, the scale is decided based on the number of integer bits and fractional bits explicitly defining the integer and decimal parts of a real number respectively. Typical HW for FXP computation is a lot simpler than its FP counterpart. In contrast, the dynamic range for FP numbers is higher than FXP as depicted in Table 3.1 for different formats. This trade-off along with a careful selection of the specific dataformats for different DNN computations has been and still remains an active research topic for achieving optimal HW-SW DNN co-design.

Apart from the previously discussed datatypes, several other explorations have also been done specifically for DNN deployments. **Binary** and **ternary** formats are at the extreme end of the datatype representations, with only 2 and 3 representable values respectively. However, even with such extreme quantizations, these highly compressed DNNs have been shown to maintain good performance, albeit at the cost of expensive and time-consuming training

Table 3.1: Comparison of different datatype metrics. [60]

Datatype	Format (e,m)	Range (dB)	Precision (ϵ)
FP32	(8,23)	1668	2^{-24}
FP16	(5,10)	241	2^{-11}
Bfloat16 [97]	(8,7)	1529	2^{-8}
HFP8 [207]	(4,3)/(5,2)	108/185	$2^{-4}/2^{-3}$
INT32	(0,31)	186.64	2^{-32}
INT8	(0,7)	42.10	2^{-8}
INT4	(0,3)	18.06	2^{-4}

procedures. Other non-conventional datatypes explored are **logarithmic** and **posit** number systems [3] which however also have more complicated HW units. Recently, a very popular datatype that have shown tight HW-SW co-optimizations are **block-floating point** (BFP) formats. As the name implies, these formats are defined as a block or a group of fixed-point numbers sharing the same exponent. They originated as a hybrid representation of FP and FXP formats complementing each other: High dynamic range of FP and efficient computations of FXP [161]. A super-set of BFP formats, where block operands can be non-FXP as well was recently proposed as a potential standard datatype for AI called **microscaling formats** (MX) [160] or block-scaled numbers as illustrated in Fig. 3.2. We discuss these formats more extensively in section 3.3.

3.2.2 Quantization methodology

While we previously discussed the different datatypes that could be used for DNN quantization, we discuss here the methodology to include these quantizations for DNN deployments. There are multiple ways of implementing quantization to a DNN and selecting the appropriate configuration could be crucial to not degrade algorithmic performance.

As was defined earlier, quantization refers to the process of mapping high-precision (continuous) values into lower-precision values. In the context of DNN deployments, this typically refers to the conversion from floating point values to integer values. Two major types of quantization functions exist – uniform and non-uniform, which are categorized based on whether the mapping is based on evenly-spaced intervals or not. The most commonly used type due to its implementation simplicity is the uniform quantization which can be defined mathematically by Eq. (3.5).

$$x_q = Q(x) = \text{clamp}\left(\left\lfloor \frac{x - z_x}{s_x} \right\rceil,\ q_{\min},\ q_{\max}\right) \tag{3.5}$$

where, x_q is the approximated quantized value of the real number x, z_x is the zero-point that offsets the values before quantization suitable for skewed distributions, s_x is the mapping scale-factor, $\lfloor \cdot \rceil$ is a rounding to nearest integer operator, $q_{\min}$ and $q_{\max}$ denote the minimum and maximum representable integers respectively, and lastly, the "clamp" operation clips the mapped integers into the desired range. $q_{\min}$ and $q_{\max}$ mainly depends on the quantization datatype which inherently including the number of bits (n_b), for ex.: signed INT8 – $(q_{\min},\ q_{\max}) = (-128, 127)$, while unsigned INT4 – $(q_{\min},\ q_{\max}) = (0, 15)$, etc. To obtain back the floating-point equivalent ($\hat{x}$) of the quantized integer (x_q), we can use the following dequantization operator: $\hat{x} = x_q \cdot s_x + z_x$.

The major challenge in effective quantization is deciding optimal scale factor (s_x) and zero-point (z_x). A special scenario of z_x=0 is called symmetric quantization, while the alternative is asymmetric quantization. Symmetric quantization is more HW-efficient for DNN computations, while more accurate representations might demand asymmetric quantizations presenting a trade-off choice [66]. There are multiple implementations of deciding a scale-factor, with some works focusing on power-of-2 factors that are more HW-efficient also called dyadic scaling. The most trivial scale calibration procedure is to match the real number range to the quantized number range. Equations 3.6 and 3.7 presents the equations for these factors considering signed and unsigned INT8 datatypes.

$$x_s = \frac{2 \cdot \max(|x|)}{128 - (-127)}, \qquad z_x = 0 \qquad \text{Symmetric quantization} \tag{3.6}$$

$$x_s = \frac{x_{\max} - x_{\min}}{(255 - 0)}, \qquad z_x = x_{\min} \qquad \text{Asymmetric quantization} \tag{3.7}$$

While such a quantization scheme is straightforward, it is often not robust to outliers leading to ineffective usage of the quantization range. To overcome this, some techniques use calibration procedures for selecting range that minimizes the quantization error based on MSE [36], KL-divergence error [94], etc. Some works use different granularities of quantization to alleviate the impacts of outliers [43]. Several other low-overhead techniques called post-training quantization (PTQ) have been explored specifically for efficient quantized inferences [66, 144]. PTQ methods are often sufficient to reach good accuracies for higher precisions and relatively easily tasks/architecture configurations. For

more challenging scenarios, quantization-aware training (QAT) can be used which performs training with simulated quantizations in forward pass through fake-quantizers [33, 56]. While quantization operation is a non-differentiable operation, it is often approximated by a straight through estimator (STE) in the backward pass [239].

The previous discussions have mainly focused on attaining efficient inference deployments, while not on reducing training efforts. Quantized training aims at reducing the complexity of performing training as well, by quantizing not only the weights and activations in the forward pass, but also the parameters (gradients and errors) encountered during the backward pass. It has also been studied that the distributions are different for backward gradients, compared to forward parameters often requiring higher dynamic range [207]. Furthermore, extensive calibration can be performed for inference scenarios, but they are not reliable for training as we get the data in the field. In such scenarios, dynamic quantization is preferred over static quantization, as we can perform runtime calibrations. Additionally, there are some analytical performance bounds that are derived for low-precision DNN performances [190], however, most of the explorations are based on extensive empirical evaluations for different quantization configurations.

3.3 Block-scaled data formats

Different datatypes along with quantization methodologies are extensively explored to push the boundaries of HW-SW co-design. That is, for a given DNN task, the aim is to maintain close to baseline accuracies while utilizing low-precision compute as much as possible. Block-scaled datatypes have been shown to achieve this very effectively as previously described in section 3.2. However, BFP as a concept is not new and has been analyzed since the 1970s [161], with the main intention of optimized deployments for DSP applications. The major focus for DSP applications was to accelerate the computations, by replacing FP operations with FXP operations, even if the storage was itself in FP formats [211]. This differs with the DNN workloads, as data transfer costs becomes critical as well with huge model sizes and frequent data transfers. With this motivation, BFP formats for DNN workloads have been attractive also as only a compression or storage format [175].

One of the major motivations for BFP was to reduce the amount of FP operations replaced by FXP operations within each block. BFP dot-product computation with vectors $\overrightarrow{x_0}, \overrightarrow{x_1}$ of block size B, common scales – e_0, e_1, integers/FXPs – $m_{0,i}, m_{1,i}$ for $i = (0, 1, \ldots, B-1)$ is as shown in Eqs. (3.8)-(3.9). The equations

showcase efficient FXP computation within a block, and enhanced dynamic range through separate exponent handling.

$$\overrightarrow{x_j} = 2^{e_j} \left[m_{j,0}, m_{j,1}, \ldots, m_{j,B-1} \right] \text{ for } j = (0,1) \tag{3.8}$$

$$\overrightarrow{x_0} \cdot \overrightarrow{x_1}^T = 2^{e_0+e_1} \sum_{i=0}^{B-1} (m_{0,i} * m_{1,i}) \tag{3.9}$$

A typical BFP computation pipeline involves converting the previous block accumulations into an FP equivalent using BFP-to-FP converter and perform summation in FP across different blocks. Lastly, the final output accumulations are converted back to BFP formats through an FP-to-BFP converter. This converter is essentially a BFP quantization operator, similar to the previously described uniform symmetric quantization (Eq. (3.6)) with power-of-2 scales and a block-level granularity, described by Eq. (3.11).

$$x_o = [2^{e_0} m_0, 2^{e_1} m_1, \ldots, 2^{e_{B-1}} m_{B-1}] \qquad m_i \text{ is a 23-bit mantissa} \tag{3.10}$$

$$e_c \text{ is } \max(e_0, e_1, e_2, ...e_{B-1})$$

$$x_{BFP} = 2^{e_c} \left[m'_0, m'_1, \ldots, m'_{B-1} \right] \qquad m'_i = \text{truncate}(m_i / 2^{e_c - e_i}) \tag{3.11}$$

It should be noted that suitable block selections by tiling matrix/tensors need to be done to enable efficient dot products as part of a larger DNN computation. Additionally, different parameters such as block size and precision also need to be chosen appropriately as they induce different quantization errors. While there are few works that derive analytical performance bounds for block size and precision [202], the specific performance for DNN computations is usually simulated and tested for different feasible combinations.

Custom BFP accelerators have demonstrated that very efficient DNN inferences [45] and also DNN training [118, 246] are achievable. Compared to other formats, lower-precision DNNs can be more easily prepared for deployment, often not needing any QAT or with only 1-2 epochs of fine-tuning [45]. These formats are especially useful for DNN training workloads as they perform good approximation for distributions with varied dynamic ranges. Multiple-level scaling across different granularities have also demonstrated similar performances at even lower precisions [43, 46]. The more general block-scaled formats of MX type propose further flexibility of block operands themselves being FP or FXP formats [47, 102]. The computation pipeline for MX formats could

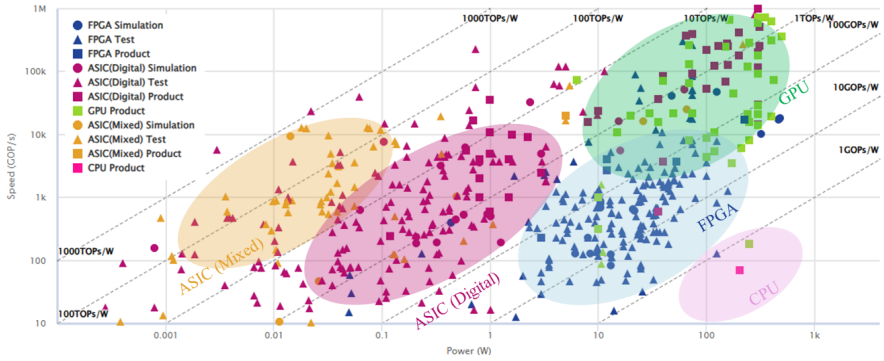


Figure 3.3: Overview of DNN processor platforms. With y-axis representing speed / performance, x-axis denoting the power consumption, and diagonal lines marking energy efficiencies. Taken from [75].

be very similar to the BFP counterparts, provided that the specific block-operand dot product and conversion units are available. Recent open-source HW implementations of the aforementioned dot product units [191] combined with the proposition of standardization for such formats [160] will potentially lead to more implementations pushing further the boundaries of HW-SW co-optimization.

3.4 Programmable vector instruction set processors for DNNs

While the previous sections of this chapter extensively described the algorithmic optimizations for DNN deployment such as quantization datatypes and procedures, we did not yet discuss the potential benefits that could be attained from a HW perspective. As we saw in section 1.1.2, multiple categories of HW exist with trade-offs of flexibility and efficiency. General-purpose categories such as CPUs and GPUs provide high programmability and flexibility, while in contrast, FPGAs and ASIPs/ASICs deliver high efficiencies as depicted in Fig. 3.3. DNNs are typically embarrassingly parallel workloads, hence a typical strategy used to increase performance has been by achieving data-level parallelism (DLP) by using SIMD/vector instructions. In this work, we mainly explore such type of processors with vector ISAs or domain-specific ISA extensions, which maintain programmability and increase performance through DLP. While these are less efficient than fixed-dataflow ASIC implementations,

they generally exhibit smaller design time and maintain compatibility for a rapidly evolving algorithm domain. Such domain-specific extensions also reduce the non-recurring engineering costs compared to ASICs [24, 87], while improving efficiencies compared to FPGAs and GPUs, hence offering an attractive middle-ground.

In the rest of this section, we firstly follow from the prior discussions on algorithmic quantization techniques and look at the potential HW benefits due to quantizing the DNNs. Then we discuss the target processors which have been explored in the thesis: general-purpose and domain-specific processors with vector instruction sets.

- **Advantages of quantized DNN**

The energy and area costs for different arithmetic operations (datatype and bitwidth) in a 45nm technology node are reported in Table 3.2, along with the memory access energies [66, 87]. In general, multiplies are costlier than additions and higher precisions are costlier than lower precision. Additionally, FP operations are more computationally demanding than integer operations. Lastly, the energy costs for memory accesses are orders of magnitude higher than arithmetic operations, especially for off-chip DRAM access. With quantized DNNs, we not only gain from lower-precision computations, but also from lower data transfer costs. For ex., instead of loading FP32 weight parameters, we could instead load only quantized INT8 weights, leading to 4× smaller data transfer size.

Additionally, the energy costs for an arithmetic operation compared to the total instruction cost in a programmable processor are very low ($< 10\%$) [87]. Hence, SIMD processing where the same instruction can be applied to multiple data greatly amortizes this programmability cost. For the same SIMD/vector length on HW, quantization with lower-precision numbers would enable increased sub-word parallelism. Therefore, quantized DNNs require fewer instructions to be processed, in turn increasing the energy-efficiency. However, extreme quantization can adversely lead to algorithmic accuracy degradation motivating the need for careful co-optimization.

- **General-purpose vector instruction-set processors**

Most popular general-purpose CPU processors for DL at the edge are based on ARM and RISC-V architectures. Also, for each different vector extensions have been released. ARM NEON is one of the most common SIMD extensions found in mobile and embedded devices such as Raspberry Pi, which have also successfully provided performance boost for many application scenarios. However, these instruction sets are for a fixed vector lengths of 128 bits. To overcome this limitation, ARM released an extension called ARM SVE which is

Table 3.2: Energy and area costs in 45nm technology node. Taken from [66, 87].

Operation	Energy (pJ)	Area (μm^2)
8b INT Add	0.03 (1.0x)	36 (1.0x)
16b INT Add	0.05 (1.7x)	67 (1.9x)
32b INT Add	0.1 (3.3x)	137 (3.8x)
8b INT Mult	0.2 (6.7x)	282 (7.8x)
32b INT Mult	3.1 (103.3x)	3495 (97.1x)
16b FP Add	0.4 (13.3x)	1360 (37.8x)
32b FP Add	0.9 (30.0x)	4184 (116.2x)
16b FP Mult	1.1 (36.7x)	1640 (45.6x)
32b FP Mult	3.7 (123.3x)	7700 (213.9x)
32b SRAM Read (8kB)	5.0 (166.7x)	N/A
32b DRAM Read	640 (21333.3x)	N/A

a scalable vector instruction set. It is based on predicated SIMD operations and its programming models are vector-length agnostic (VLA) providing high levels of HW deployment flexibility and SW portability. Through its predication support it enables efficient control of different lanes of vector instruction up to a granularity of 8-bits and also enables efficient loop tail handling.

Table 3.3 tabulates few of the already commercially available processors with ARM SVE support. It details different configurations, but what is important to notice is that similar ISA and in turn the same programming model can be used at different application levels with differing vector lengths: data-center or server level, or mobile processors and potentially also MCUs in the future. While larger vector lengths enable higher DLP and are preferred for DNN computations, they might not be optimally used for all general-purpose applications. This could also be the reason for the interesting choice of reducing the vector length for a later generation, from 256-bits in Neoverse V1 to 128-bits in Neoverse V2. However, no concrete reason has been given for this to the best of our knowledge. Another reason could be to maintain compatibility with the previous SW library supports of ARM NEON extension. There will potentially be more commercially available supported devices in the near future enabling a wide-spread impact and easy usability of SVE DNN kernel codes. While different application domains can be supported by the same code, our main focus is on the edge or mobile processor deployments.

RISC-V is an open-source ISA has its own vector instruction set called RVV which also utilizes a VLA programming model, with its latest stable version being RVV-1.0. RVV has even more flexibility than ARM SVE supporting vector

Table 3.3: ARM-SVE supported processors.

	Fujitsu A64FX	**AWS Graviton 3 (Neoverse V1)**	**NVIDIA Grace C1 (Neoverse V2)**	**Cortex X2 (Exynos 2200)**	**Cortex A710 (Exynos 2200)**	**Cortex A510 (Exynos 2200)**
Year	2021	2022	2023	2022	2022	2022
Application domain	Supercomputer	Server	Server	Mobile	Mobile	Mobile
ISA	SVE	SVE	SVE2	SVE2	SVE2	SVE2
Architecture	v8.2-A	v8.5-A	v9.0-A	v9.0-A	v9.0-A	v9.0-A
Execution	OOE	OOE	OOE	OOE	OOE	In-order
Vector Length (bits)	512	256	128	128	128	128
# Cores	48	64	72	1	3	4
L1 Cache per core (I$/D$)	64 KB	64 KB	64 KB	64 KB	32-64 KB	32-64 KB
L2 Cache	32 MB	1 MB/core	1 MB/core	1 MB/core	512 KB	512 KB
L3 Cache	N/A	32 MB	114 MB	8 MB	-	-

lengths up to 16384 bits, inspired from the cray-style vector processing [189] as opposed to a packed/predicated SIMD processing. RVV has few other key differences compared to ARM SVE, one of the main ones being the feature of register grouping. Register grouping in RVV enables working on data vectors which are smaller / larger than the actual HW vector lengths. Even though ARM-SVE is also VLA-based programming model, the actual runtime flexibility is based on the maximum HW vector length (i.e., runtime deployable vector length configurations can be any 128-bit multiples up to a maximum of the HW vector length). While with RVV, integer multiples and fractional values of register grouping values allows the vector instruction to be applied on vector lengths larger than or smaller than the HW vector length. However, how well the register grouping could be used is implementation-dependent and needs to be explored [25, 171].

- **Domain-specific vector instruction-set processors**

For more efficient implementations, we target a domain-specific instruction-set processor [242] with SoftSIMD instructions. The details of the HW and quantized DNN experimentation with these instructions are discussed in Chapter 8. The key concept which is different from the previous SIMD instruction set (HardSIMD) is illustrated in Fig. 3.4. The major characteristics of SoftSIMD processor are as follows:

- Capability of performing sub-byte vector instructions (3/4/6/8-bits) with high flexibility and full utilization of vector lengths (integer multiples of 48-bit).
- Supported vector instructions: Vector addition/subtraction/shift and vector by scalar multiplication.

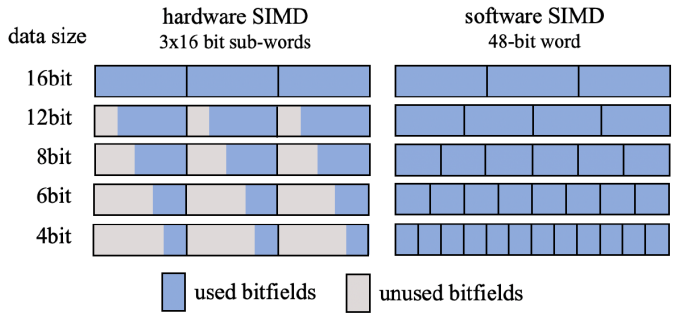


Figure 3.4: SoftSIMD vs HardSIMD HW. [240]

- Repacking operations to convert across different precisions seamlessly, which is used effectively to avoid overflows.

SoftSIMD achieves this flexibility through a bit-serial computation approach of multiple add and shift instructions. This allows the reduction of critical path significantly compared to HardSIMD multiplier units. Additionally, the performance can be further increased through dynamic frequency scaling based on the actual runtime bit-width configurations [241].

3.5 Conclusion

This chapter explored some of the most prominent algorithmic optimizations that have been performed for efficient DNNs. We delved deeper into one of the most widely-used techniques–quantization, for effective DNN deployments on HW, utilizing the fact that DNN performance does not mandate high-precision computations everywhere.

We specifically covered the methodology for block-quantization, which involves a fine granularity of values (16-64) sharing a common power-of-2 scale. This has recently also gained a lot of traction as an efficient DNN deployment technique across diverse workloads.

Additionally, the development of optimized HW platforms, alongside algorithmic solutions, is crucial for advancing the efficiency of DL. An overview of different processors and their trade-offs for DNN deployment was detailed. Lastly, the importance of programmability with increased parallelism through vector instructions was highlighted, concluding with potential general-purpose and domain-specific processors satisfying the aforementioned objective.

Chapter 4

Hardware-aware meta-learning for quantized few-shot learning

Building upon the technical background covered in the previous chapters 2 and 3, in this chapter we explore on techniques to answer the research question defined in section 1.2: *Q1. How to efficiently deploy data-efficient deep learning algorithms on resource-constrained hardware?* In particular, we explore few-shot image classification scenarios with 1-5 training samples per class on previously unseen tasks, while the algorithms are more generally applicable to any other learning task such as regression and reinforcement learning.

With this objective, we investigate quantized training with BFP quantized weights (W) and activations (A) in the forward pass, and gradients (G) and errors (E) in the backward pass to perform training with only HW-efficient GeMM computations. We also perform different mixed-precision training explorations across WAGE parameters, with initializations from conventional pre-training approaches typically used for few-shot learning (FSL). Additionally, we identify more effective initializations for low-precision configurations through upfront inclusion of target HW constraints in the pre-training process. Utilizing our modified approach, we have obtained up to ∼90% faster convergence compared to conventional initializations while reducing the accuracy loss by a factor of ∼1.3x.

This chapter is based on the published article: Nitish Satya Murthy, Peter Vrancx, Nathan Laubeuf, Peter Debacker, Francky Catthoor, and Marian

Verhelst. *"Learn to Learn on Chip: Hardware-aware Meta-learning for Quantized Few-shot Learning at the Edge."* In 2022 IEEE/ACM 7th Symposium on Edge Computing (SEC), pp. 14-25. IEEE, 2022 [141].

4.1 Introduction

With the rise of edge computing, deploying deep neural network models (DNN) on edge devices has been a prominent field of research. While major focus has been on inference accelerator for the edge, the ability to also train these models on-chip greatly increases the scope and performance of such devices [132]. The capacity to update the networks at the edge, without sending data to the cloud, also makes these devices an ideal platform for privacy-sensitive applications, such as user-personalization [22].

However, personalization applications on edge devices introduce certain challenges in terms of training. Training data is often limited, train and test conditions can differ, computational power is limited, etc. [70]. The limited data constraints have inspired new few-shot learning (FSL) techniques for normally data-hungry DNNs, that aim to reduce overfitting and enhance generalization capabilities. FSL enables a DNN to solve new unseen tasks after seeing only a few training examples (shots). Several FSL algorithms exist, based on different techniques like metric learning [201], memory augmented neural networks [193] and optimization-based meta-learning [58].

Optimization-based meta-learning or meta-learning is associated with utilizing prior knowledge in a form of pre-training on a large corpus of few-shot train tasks. The objective then is to obtain a more suitable network initialization for later adaptation or fine-tuning on previously unseen test few-shot tasks. The algorithm is also model-agnostic, in terms of being applicable to any gradient-descent based learning algorithms [58]. We study the *Reptile* algorithm [152], one of the popular variants of meta-learning, due to its lower computational complexity and its ability to find weight initialization that lie close to other task minima [69]. Prior work has investigated and highlighted the advantages of using meta-learning for personalization applications such as keyword spotting [70], activity recognition [70], face recognition [198], biometric identification [245], etc.

While the data constraints in personalization are handled well by meta-learning, the fine-tuning process itself is not yet hardware-friendly and suffers from the memory and energy restrictions imposed by on-chip training [127]. To overcome these challenges, most of the on-chip training papers rely on quantized training [60, 231, 236]. Performing backpropagation efficiently on chip as such

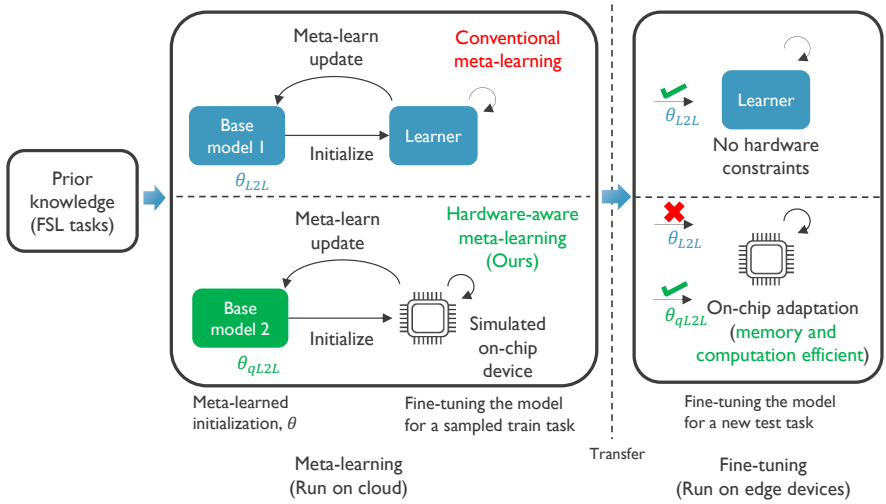


Figure 4.1: Comparison between conventional meta-learning and our modified hardware-aware meta-learning process.

involves quantizing the weights and activations in the forward pass, as well as the gradients and errors in the backward pass.

Since the initialization of networks is an important step for FSL [152] and low-precision training [236], our aim is to find better initializations that are now suitable for both few-shot as well as low-precision adaptation. In this regard, we introduce a modified version of the *Reptile* meta-learning algorithm [152] which upfront considers the hardware on which the adaptation process would take place. As can be seen from Fig. 4.1, the conventional meta-learning algorithm considers a learner without any hardware constraints. In contrast, our target is area- and energy-efficient on-chip learning, which is also simulated as the learner network of the *Reptile* algorithm to make it hardware-aware. Through extensive experiments, we demonstrate the advantages of our method to find better initializations and also highlight the hardware benefits by enabling the on-chip training of DNNs.

Our main contributions in this chapter are hence as follows:

- We propose a hardware-aware Reptile-based meta-learning technique, well-suited for few-shot adaptation on-chip.
- With this technique, we demonstrate quantized fine-tuning performance on unseen few-shot tasks using low-precision BFP datatypes in the forward

and backward passes.

- Comparing our initializations with conventional *Reptile*, we obtain up to ∼90% faster convergence while reducing the accuracy loss by a factor of ∼1.3x.
- We provide extensive ablation studies to demonstrate the individual contributions of our design choices.

The rest of the chapter is organized as follows. Section 4.2 provides the background information on the main concepts used in this chapter. Section 4.3 details the innovation and methodology of our quantized meta-learning algorithm. Sections 4.4 and 4.5 explain the experiments carried out and the ablation studies to highlight the need of making the algorithm hardware-aware. In section 4.6, we compare our approach with related works from the SoTA. Lastly, in section 4.7 we discuss the conclusions of this chapter.

4.2 Background theory

4.2.1 Learn to learn - Meta-learning (L2L)

A few-shot learning problem, as its name suggests, involves training a DNN with limited amount of training data (shots). An FSL classification problem is usually referred to as an N-way, K-shot task, which indicates the objective to classify among N classes with only K training samples per class (typically less than 10). As DNNs keep getting bigger and more complex, the chances of overfitting the model on training data become higher, especially when the data is limited. Several algorithms have been researched to solve this problem [228].

Meta-learning approaches have been a popular approach for FSL tasks with its inception from Model Agnostic Meta-Learning (MAML) [58]. MAML, through its pre-training technique, learns the model parameters for optimal transfer. It achieves this through a bi-level optimization loop, where the inner loop is the fine-tuning step on a new task after transfer, and the outer loop involves iterating over many tasks to decrease the loss due to inefficient transfer. While MAML was efficient in solving FSL tasks, it required second-order derivatives in each outer loop step which made it computationally demanding. The *Reptile* algorithm has the intent of making meta-learning computationally simpler by only using first-order updates, which becomes more crucial when fine-tuning tasks require more gradient descent steps [152].

Algorithm 1 defines the meta-learning process of the batch version of *Reptile*. Base model parameters are transferred to a learner and finding good initialization

Algorithm 1 Reptile meta-learning (conventional [152] and quantized).

1: Randomly initialize base-model parameter θ
2: **for** iteration = 1,2,...n **do**
3: Sample m tasks $\tau_1, \tau_2, ...\tau_m \sim p(T)$
4: **for** i=1,2,...m **do**
5: Compute $\theta_i^* = U_q(L_{\tau_i}, \theta, k)$ *Fine-tune learner (FP32* or quantized)
6: **end for**
7: $\theta \leftarrow \theta + \beta \frac{1}{m} \sum_{i=1}^{m} (\theta_i^* - \theta)$ *Meta-learn update*
8: **end for**

where,
- n - Outer loop iterations
- m - Number of tasks in meta-learning batch
- U_q - Perform k iterations of gradient-descent (FP32 or quantized)
- L_{τ_i} - Loss on task τ_i
- k - Inner loop iterations
- β - Outer update step size
- θ - Base model parameter
- θ^* - Learner model parameter after fine-tuning

parameters forms the optimization objective of the meta-learning process. In *Reptile*, the meta-learning step involves a simple interpolation of the initialization and the fine-tuned weights, to eventually compute initializations that lie close to other task minima. Our method involves a modification of the fine-tuning step of algorithm 1 to enable adaptation on target low-resource hardware. This modification is described in detail in section 4.3.

In the context of fine-tuning, most of the works select a transductive approach, i.e., they evaluate the complete test set at once. It has also been studied that a transductive approach almost always outperforms a non-transductive approach [18, 152]. However, in a practical scenario of on-chip fine-tuning, the objective is to process the samples on the fly and not wait for more samples. This limits the usage of transductive fine-tuning. Hence, in our studies we always perform the fine-tuning in a non-transductive manner.

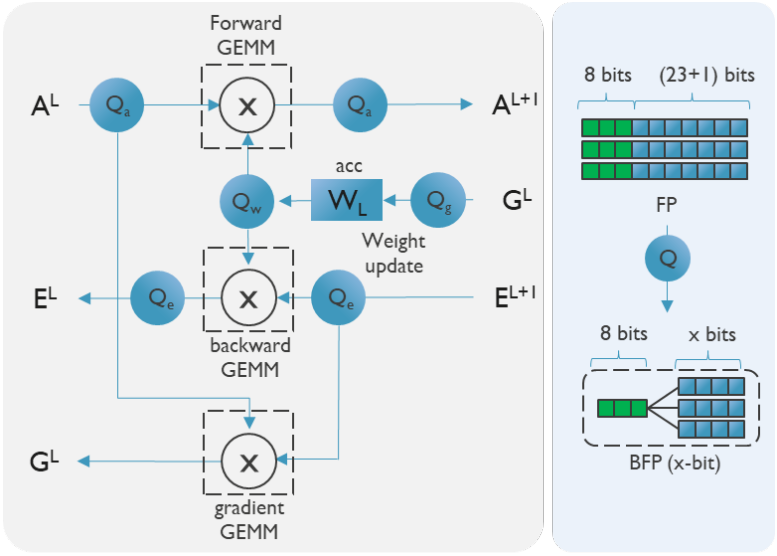


Figure 4.2: Quantized training by including fake quantizer nodes in forward and backward propagation [224] [247]. Quantizer nodes simulating BFP (x-bit).

4.2.2 Learn on chip - Quantized training

Several papers in literature have tried to reduce the computational efforts involved in training DNNs by utilizing reduced precision or quantized computing. Low-precision compute becomes attractive for on-chip training as it relaxes the memory and computational constraints. One of the key challenges of training that demands memory is that it requires storing activations of all the layers to perform a single backpropagation step. This is also studied to be the main memory bottleneck, which could be mitigated by downsampled activations storage [22], or by utilizing low-precision activations. Additionally, training is also time-consuming because of the update locking phenomenon, which is the idle time to calculate gradients until next iteration can commence [93]. Hence, reducing training iterations not only benefits the real-time adaptation experience, but also reduces the overall energy consumption.

With low-precision computation, FP / FXP / BFP, etc., mainly differ in their data representation with the same number of bits as was described in detail in section 3.2. BFP, in particular, is a combination of FP and FXP datatypes, wherein each block has a common exponent and each number within the block has separate mantissas. Although DNN training process is involved as shown in Fig. 4.2, the predominant operations are general matrix multiplications

(GeMMs). From the figure, we can also observe that the quantizer nodes are inserted in both the forward and backward passes to simulate quantized training [247], which ensures the GeMM operands are of low-precision.

4.3 Learn to learn on chip - Quantized meta-learning (qL2L)

In this section, we explore the motivations of the proposed modified meta-learning technique, elaborating on its methodology.

Motivation

Our work could be motivated in terms of hardware-aware training or from a meta-learning context of optimizing for better transfer. From a training perspective, it is known beforehand that hardware-aware methods such as QAT performs better than PTQ for quantized inference, especially for lower precision networks [145]. In our scenario, it is even more essential as we consider quantized training which is more complex than quantized inference. Additionally, we also consider limited data that limits training capabilities. Hence, performing pre-training that includes hardware constraints upfront, even though it makes the pre-training more costly, enables more effective on-chip adaptation.

In a meta-learning context, *Reptile* aims at finding initializations for quick adaptation to new few-shot tasks. It achieves this by pre-training on several similar train tasks to learn an efficient transfer. The impacts of the similarity of the train tasks is also important, which has been analyzed previously in terms of domain difference, indicating a more effective transfer when the test / train domains have higher similarity [31]. However, when the test task adaptation is to be performed on low-precision networks, it differs from the FP32 adaptation typically considered during the pre-training stage. Hence, this could be understood as a change in the distribution of tasks due to differences in train and test tasks, which reduces the efficiency of transfer. Consequently, by making sure that the task setup is similar during meta-learning and fine-tuning, we ensure that the new initialization found is more suitable for low-resource hardware scenarios.

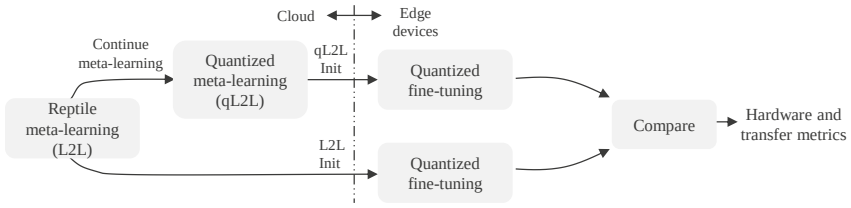


Figure 4.3: Methodology for performing quantized meta-learning (qL2L) and the difference with conventional meta-learning (L2L).

Methodology

Low-precision compute is indeed a challenging task, and to make it easier, several works have resorted to first learning an FP32 solution and then perform the more complex task of quantized learning [220]. A similar methodology is employed here as shown in Fig. 4.3, wherein the FP32 meta-learning is initially performed. Quantized fine-tuning is then done to study the performance degradation due to ineffective transfer from FP32 solution. Furthermore, it was observed empirically that this also provides insight into selecting hyperparameters in the quantized meta-learning stage as we see later in section 4.5. With this knowledge, quantized meta-learning is performed to adapt the base-model towards more hardware-efficient fine-tuning models. Our main constraints in the hardware process are the inclusion of quantization during training.

While performing quantized meta-learning, we continue algorithm 1 with the FP32 meta-learned solution, instead of a random initialization. The gradient-descent process of $U_q(L_{\tau_i}, \theta, k)$ now involves performing quantized training for k steps with initialization θ on quantized network loss L_{τ_i}. It is also to be noted that only the inner loop part of meta-learning is quantized and the outer loop meta-learning update is not, as the intention is to emulate only the low-precision fine-tuning in the meta-learning process. The impact and setting of qL2L hyperparameters such as inner loop (k) and outer loop iterations (n) will be explored in section 4.5.

Quantization

Following the discussion from section 3.2 and 4.2 on BFP advantages, here we provide more details regarding the BFP quantization used in this chapter. This quantization is included in the DNN training graph for weights (W) and

activations (A) in the forward, and gradients (G) and errors (E) in the backward pass [247].

The configurations used in our work is referred to as BFP-m where each block has m bits mantissa precision (no implicit leading 1). It is also to be noted that the rounding is done stochastically, which has been studied to reduce the quantization errors during training [77, 236]. Regarding the granularity of block sizes, there exists a trade-off of hardware efficiency vs algorithmic performance. With increasing block sizes, we obtain denser and more efficient compute, but at the cost of accuracy degradation [45]. We explored two different block sizes, one at the complete tensor level and a smaller one (per filter for weights, per image for activations). It was observed that the performance severely decays at tensor-level block sizes, hence, we proceed with smaller blocks throughout our experiments. All experiments are run on NVIDIA A100 GPUs by simulating quantization using libraries in the PyTorch framework [166, 247]. We target on-chip devices with BFP datatypes with low-precision mantissa bits (4-bits) for GeMM operands.

4.4 Experimental setup and results

Dataset: In our study, we consider Omniglot [113], one of the standard FSL benchmarking datasets. It consists of 1623 samples of 50 handwritten characters of size $28 \times 28 \times 1$. Similar to the original *Reptile* study, we consider 4 possible combinations of ways (5,20) and shots (1,5).

Model architecture: A conv4 architecture, as the one predominantly chosen in prior meta-learning study is used here [58, 152]. It is essentially 4 convolutional blocks followed by a linear layer for classification. Each convolutional block contains a 2D convolution with 64 3×3 filters, followed by batch norm and then a ReLU activation.

Evaluation metrics: Our main innovation is in the modification of the conventional meta-learning approach. Hence, the metrics should be relevant to compare the efficiency in transferring different initializations. We refer to some standard metrics for evaluating transfer: jumpstart, asymptotic performance and time to threshold [214]. However, in this scenario the initialization is not intended for inference, rather to perform suitable fine-tuning from that. Hence, only last two metrics are of relevance.
Asymptotic accuracy, as the name suggests, represents the maximum accuracy reached. Δ accuracy is used for comparison between L2L and qL2L

Table 4.1: Different experimental settings for analyzing quantized meta-learning.

Sl. No.	Scenario	Meta-learning	Fine-tuning	Low-resource fine-tuning hardware?	Effective transfer?
1	Reptile for FP32 fine-tuning	L2L – FP32 [152]	FP32 fine-tuning [152]	No	Yes
2	Reptile for quantized fine-tuning	L2L – FP32 [152]	Quantized (BFP) fine-tuning (This work)	Yes	No
3	Quantized meta-learning for quantized fine-tuning	qL2L – BFP (This work)	Quantized (BFP) fine-tuning (This work)	Yes	Yes

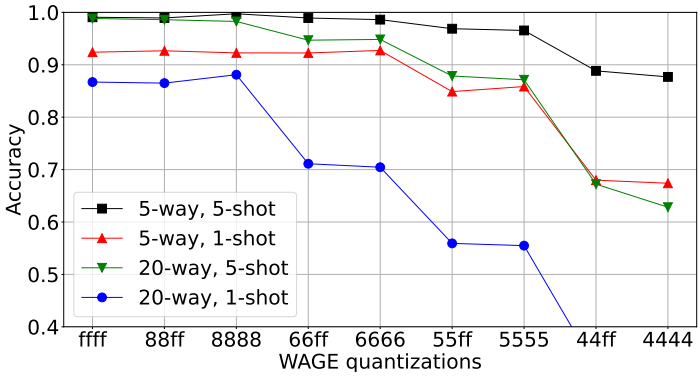


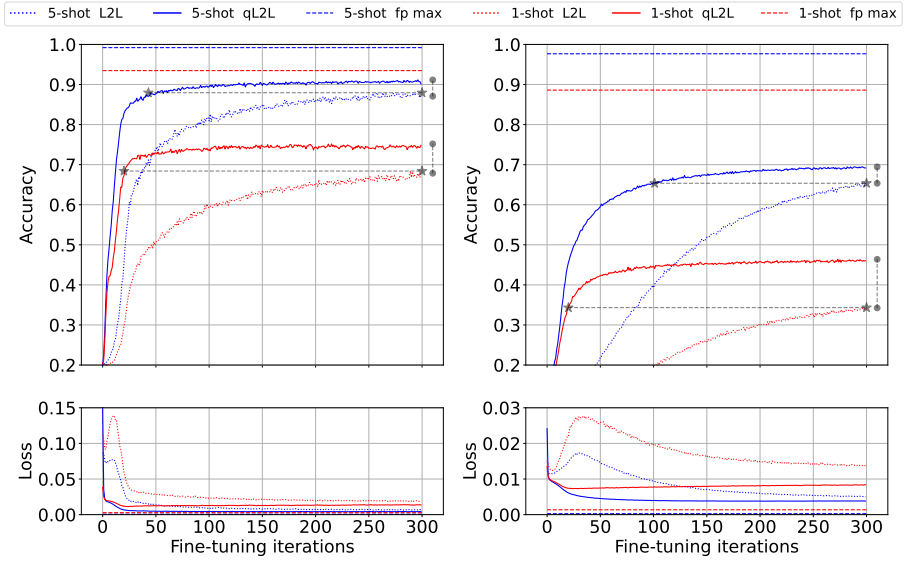
Figure 4.4: Asymptotic accuracies after fine-tuning with different BFP quantizations using L2L initialization. (WAGE denotes Weights, Activations, Gradients and Errors. f represents FP32).

initializations.
Time to threshold compares the time difference between the two trainings to reach certain threshold accuracy, which in our case, we define as the % difference in speed to reach asymptotic accuracy of L2L initialization.
We also consider the following hardware metrics to compare ease of adaptation – model size and activation memory for one full batch gradient descent step.

Baseline comparisons and results discussion:
We consider 3 scenarios for comparison as shown in Table 4.1. The first scenario involves using the conventional *Reptile* algorithm (L2L) to perform FP32 fine-tuning on new few-shot tasks [152]. While this provides the highest accuracy, the intended fine-tuning is also in FP32 and hence not feasible for deployment on

edge devices. However, the transfer is still effective because of similar hardware constraints in meta-learning and fine-tuning.

In the next scenario, using the previous L2L initialization, we analyze the performance of quantized fine-tuning. This simulates the case where we deploy the networks on a limited resource edge device, but do not account for this in the meta-learning process. Plots in Fig. 4.4 show the network sensitivity to different quantization settings, with the same L2L initialization. The first obvious trend which is seen is the decrease in accuracies compared to FP32 baseline as we go towards lower precision datatypes such as BFP4. The more interesting observation is that the quantizations of gradients and errors do not affect the performance much between FP32 and a lower precision. This is in stark contrast to non-FSL training scenarios, which train from scratch, where the quantizations in the backward pass demand more careful considerations [207]. While these network adaptations are well-suited for low-resource hardware, they incur severe performance degradation due to ineffective transfer from FP32 meta-learning. Fine-tuning curves from this L2L initialization for BFP-4 quantizations can be seen in Fig. 4.5. It should be noted that these curves are an average of 200 test FSL tasks performed in a non-transductive manner for 300 iterations. The baseline FP32 fine-tuning accuracies are also plotted for comparison. The loss



(a) 5-way classification. **(b)** 20-way classification.

Figure 4.5: Fine-tuning curves for Omniglot classification.

Table 4.2: Fine-tuning comparison of Omniglot (L2L vs qL2L initializations).

Ways	Shots	WAGE (quantization bits)	Accuracy		Asymptotic accuracy Δ (%)	Convergence speed Δ (%)	Model size (MB)	Activation memory per sample (MB)
			L2L	qL2L				
5	5	FP	99.20	-	-	-	0.450 (8×)	0.136 (8×)
		4444	87.94	91.15	3.20	85.66	0.056 (1×)	0.017 (1×)
5	1	FP	93.46	-	-	-	0.450 (8×)	0.136 (8×)
		4444	68.40	75.18	6.78	93.33	0.056 (1×)	0.017 (1×)
20	5	FP	97.66	-	-	-	0.450 (8×)	0.136 (8×)
		4444	65.35	69.62	4.26	66.33	0.056 (1×)	0.017 (1×)
20	1	FP	88.60	-	-	-	0.450 (8×)	0.136 (8×)
		4444	36.32	46.38	10.05	93.33	0.056 (1×)	0.017 (1×)

curve in particular, provides further insight into the optimization landscape around the L2L initialization. The increasing loss in initial iterations reiterates the fact that L2L is a suboptimal initialization for quantized fine-tuning.

Lastly, hardware-aware or quantized meta-learning (qL2L) is performed to obtain effective transfer for quantized fine-tuning. The hyperparameter settings followed in these experiments are listed in Appendix A. It should further be noted that the learning rates while fine-tuning were not modified to ensure fair comparisons between initializations. The choice and impact of other relevant hyperparameter settings are discussed in detail in section 4.5.

In Fig. 4.5, the two quantized fine-tuning curves for the same test tasks and for various combinations of FSL classifications are plotted. The smooth and decreasing loss curves with qL2L initialization confirms that qL2L is better than L2L for the objective of quantized fine-tuning. It can also be seen that qL2L transfer yields much better fine-tuning results in both transfer learning metrics: time to threshold (denoted with stars) and asymptotic accuracy (denoted with circles). These metrics are also reported in a tabular manner in Table 4.2. 20-way, 1-shot classification yields as high as 10% greater accuracies and 90% faster convergence. Along with the transfer metrics, we also compare the batch activation memory and model size of the final models. Although there is accuracy degradation compared to FP32 even with qL2L, we can also see that 8 times lower memory is required for the quantized models.

4.5 Analysis

In this section, we perform further analyses by performing ablation studies to answer the following questions:

- How to select hyperparameters (inner / outer loop iterations) for qL2L and analyze its impacts on fine-tuning?
- What are the effects of different fine-tuning with the same initializations?

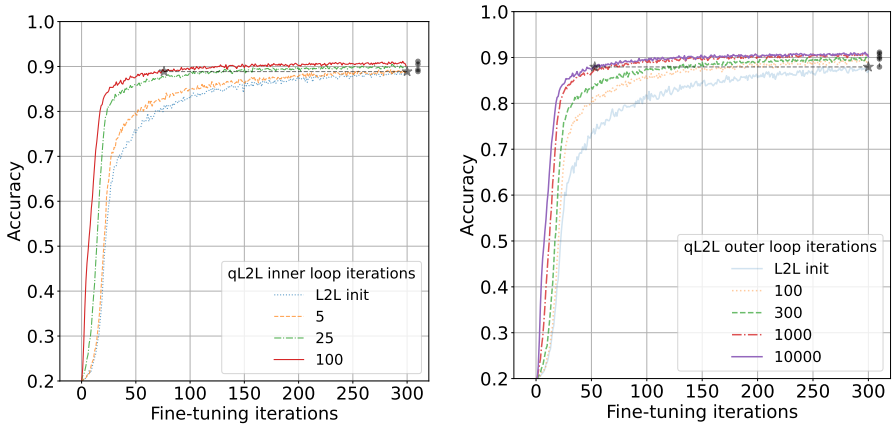
4.5.1 Effect of hyperparameters during quantized meta-learning

Inner loop iterations in qL2L

As we previously saw in Fig. 4.5 in the graphs of Omniglot fine-tuning, the time to reach convergence starting from L2L initialization was longer for quantized fine-tuning. The original settings of *Reptile* for inner loop iterations is 5/10, but these iterations are not enough to get good performance for quantized fine-tuning. Hence, a higher value is needed in order to reach task minima for each sampled task during qL2L. As can be seen from Fig. 4.6a, we get increasingly better performance by increasing the qL2L inner loop iterations. Furthermore, it is also shown in Fig. 4.7 that the same hyperparameters when applied to L2L does not yield much benefit as compared to qL2L. This reaffirms the advantages of the upfront inclusion of quantization during meta-learning.

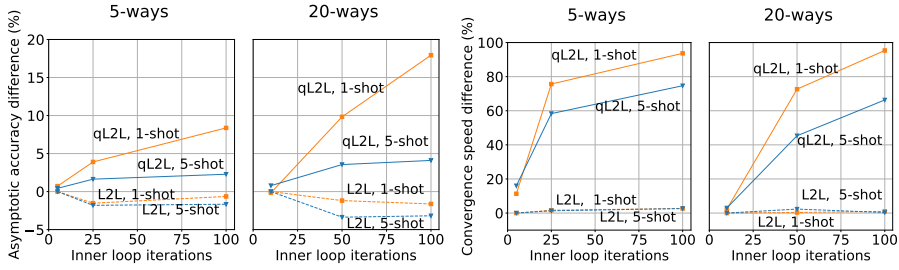
Outer loop iterations in qL2L

Outer loop iterations is another such hyperparameter which needs to be set carefully. Essentially, as the number of iterations increase, the initialization moves farther away from that of conventional reptile and towards the one where quantized fine-tuning is desirable. This can also be easily inferred from Fig. 4.6b, as fine-tuning curves improves with more qL2L outer loop iterations. Furthermore, in the initial iterations of quantized fine-tuning in Fig. 4.8, we observe that the loss increases with lower qL2L outer loop iterations. As we start from a better initialization point, that is more hardware-aware, we do indeed observe better convergence in the loss curves. This also gives insight into the amount of qL2L outer loop iterations required, a threshold beyond which we do not observe many benefits. Such a value, for example 300 for 5-way 5-shot classification is much lower than the initial L2L meta-learning iterations of 100,000 indicating only small overheads of performing quantized meta-learning.



(a) Different inner loop iterations. **(b)** Different outer loop iterations.

Figure 4.6: Fine-tuning plots for qL2L (5-way, 5-shot).

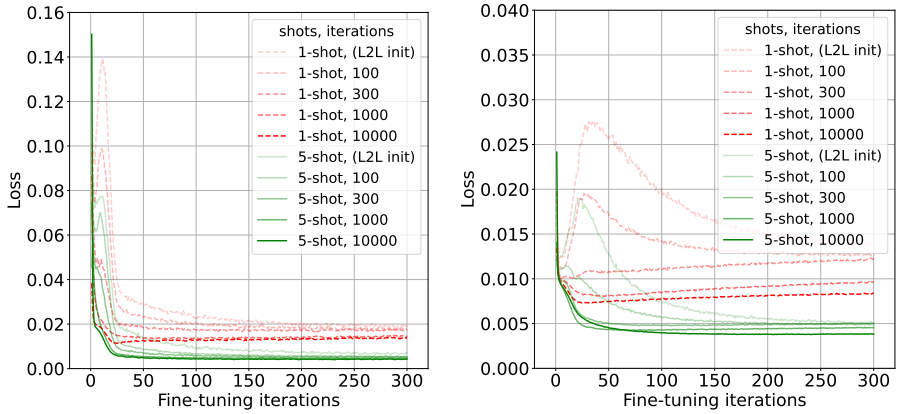


(a) Accuracy difference. **(b)** Convergence speed difference.

Figure 4.7: Performance comparisons for different inner loop iterations in L2L and qL2L.

4.5.2 Effect of different fine-tuning with same initializations

An ablation study we also explore is FP32 fine-tuning with a qL2L initialization. The difference in performances compared to L2L would also imply that we are in different optimization landscapes. In Fig. 4.9, we plot the fine-tuning curves of all 4 possible combinations: Conventional or quantized meta-learning, full-precision or quantized fine-tuning. And indeed, we can observe the differences between L2L and qL2L initializations. L2L provides good full-precision and poor quantized fine-tuning performances. With a hardware-aware qL2L initialization, although we obtain better quantized fine-tuning, we observe worse performance



(a) 5-way classification. **(b)** 20-way classification.

Figure 4.8: Fine-tuning loss curves for different outer loop iterations in qL2L.

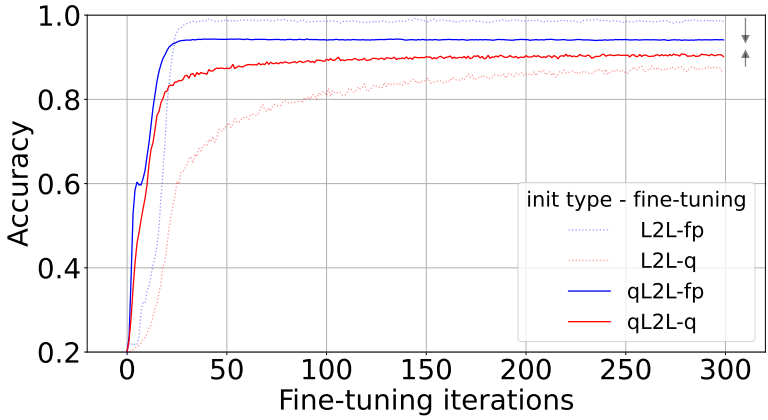


Figure 4.9: Fine-tuning plots with all combinations of L2L/qL2L initializations and FP32/BFP quantized fine-tuning (5-way, 5-shot).

when adapting the NNs at higher precisions. This reaffirms that with qL2L we obtain hardware-aware initializations that are more suitable specifically for quantized fine-tuning.

Table 4.3: Different approaches for NN adaptation on hardware.

FSL Technique	Refs	On-chip Training?	Non-classification Tasks?	Quantization bit-widths				Optimized For?
				W	A	G	E	
Non-FSL	[231] [207] [208]	Yes	Yes	2 8 4	8 8 4	8 8 8	8 8 4	Quantized training from scratch
Metric Learning	[99], [182], [180], [112], [151]	No	No	FP	FP	FP	FP	Quantized similarity evaluation
Optimization-based (MAML, Reptile, Meta-SGD)	[223]	Yes	Yes	16	FP	16	FP	Quantized meta-learning
Optimization-based (MAML)	[119]	Yes	Yes	4	4	8	4	Efficient quantized fine-tuning
Optimization-based (Reptile)	**This work**	**Yes**	**Yes**	**4**	**4**	**4**	**4**	**Effective transfer for quantized fine-tuning**

4.6 Related work

Table 4.3 tabulates the different works in literature related to training NNs on hardware efficient platforms. We partition this table initially based on their application area: whether training from scratch with large data or for FSL applications i.e. adapting from limited data. We will also detail other information such as applicability, WAGE quantizations and main objectives of the works for better comparisons.

The first category is the non-FSL based approaches, where the works mostly rely on different datatypes for NN parameters based on their sensitivities during training: [231] uses integers for efficient computing, [207] utilize hybrid 8-bit FP with different configurations in forward and backward passes to handle varying dynamic range requirements, [208] uses unique 4-bit FPs with radix-4 to capture wider ranges with lower bit widths. However, the works discussed in this subcategory aim at more efficient NN training from scratch without any consideration of limited data constraint as in few-shot learning scenarios.

The next categories are related to works that mainly target FSL applications, or scenarios with only limited training data on hardware-efficient platforms. One

of the FSL subcategories relies on using the NN as an embedding function to either generate embeddings for metric learning or addresses to query a memory which returns class labels. These works have showcased efficient performance for scaled FSL tasks with as high as 100 classes and also require no fine-tuning of the NNs after deployment. However, they mainly consider non-quantized NN and focus only on optimizing the evaluation of similarity metrics on hardware. Furthermore, [182] calculates the breakdown of system computation and shows that the NN computation is more demanding than the similarity metric, which becomes more prominent when the task is smaller. Hence, optimizing the whole system end-to-end including the NN becomes important. Lastly, these FSL approaches rely on similarities of different classes and hence are mostly limited to classification tasks. On the other hand, many non-classification tasks also benefit from data-efficient learning. For example, the number of samples required to enable continuous control of robotics systems is important, and has also shown to benefit from FSL [58].

In the optimization-based meta-learning category, only few papers focus on approaching this while also targeting low-resource hardware constraint. One of the work's main objective was to reduce the meta-learning complexity on edge devices [223]. However, they do not analyze the impacts on fine-tuning extensively. Additionally, there is no mention of quantizations of activations being considered in their approach, which is crucial for performing memory-efficient training on edge devices [22]. Lastly, they do not experiment on low-precisions and mainly consider 16-bit integers in their studies.

Another work in this subcategory closely matches our objective of NN adaptation to previously unseen tasks with limited data on-chip [119]. They use low precisions for training operations and are also able to reach accuracies close to baseline performance of MAML models. Their quantization framework learns quantization parameters for NN adaptation by observing the parameter statistics during meta-learning, MAML specifically. Although 8-bit gradients are used, the memory requirements during training are relaxed by storing only parts of the weight in high-precision. While they have good performance, their investigations focus on finding efficient quantization settings, which deviates from our analysis of finding initializations to transfer effectively for target quantized fine-tuning. It is also unclear if they consider non-transductive fine-tuning in their works similar to our configurations. However, their techniques are complementary to the work presented here and could be potentially combined with ours to further increase performance.

4.7 Conclusions

In this chapter, our objective was to enable efficient few-shot learning for image classification modalities in combination with block-scaled quantizations for achieving deployment efficiencies.

Towards this goal, we propose a modified meta-learning technique that is hardware-aware, to enable quantized fine-tuning of unseen tasks with limited data. We make use of block floating point datatypes during network adaptation. The computations are quantized end-to-end, all the way down to 4-bit matrix-multiplication operands, realizing a hardware-efficient training pipeline on-chip. Through extensive experiments on Omniglot datasets, we showcase that the initialization learned by our technique transfers more effectively for adapting on low-resource hardware, for instance, reducing the accuracy loss by a factor of $\sim$1.3x and converging around 90% faster in 20-way scenarios.

The next chapter explores a similar objective of deploying data-efficient DL algorithms on resource-constrained hardware, however, focuses on a different modality – continuous control tasks. These tasks are often performed through RL algorithms, demanding several interactions with the environment, while data-efficient RL algorithms reduce this burden considerably. This would provide insights on efficiently enabling futuristic autonomous control and robotics systems.

Chapter 5

Hardware-efficient planning in model-based RL for continuous control tasks

This chapter explores methods addressing the research question defined in section 1.2: *Q1. How to efficiently deploy data-efficient deep learning algorithms on resource-constrained hardware* for continuous control tasks?

Reinforcement Learning (RL) algorithms have been the popular choice for targeting continuous control tasks, due to their autonomous decision-making capabilities [209]. For achieving task-generalization and increasing sample-efficiency, Model-based RL (MBRL) algorithms in particular have shown tremendous potential compared to Model-free RL (MFRL) counterparts as also highlighted in Fig. 5.1. These MBRL algorithms typically involve a dynamics model (DNN-based), learning the state-transitions of the target system in an environment, which is then utilized for planning and control [38]. In this chapter, we explore block-quantization for the aforementioned dynamics models targeting HW-efficient deployments. Conventional MBRL algorithms even without any low-precision quantization often encounter the challenge of compounding-error, where any model error propagates and compounds along the prediction horizon due to multi-sequence inferences [95]. Quantization of DNN-based dynamics models can further increase this error, degrading control performance. Hence, these tasks require greater attention during quantization compared to conventional DNNs involving single-shot inferences. We optimize the quantization configurations by analyzing its performance impacts on the

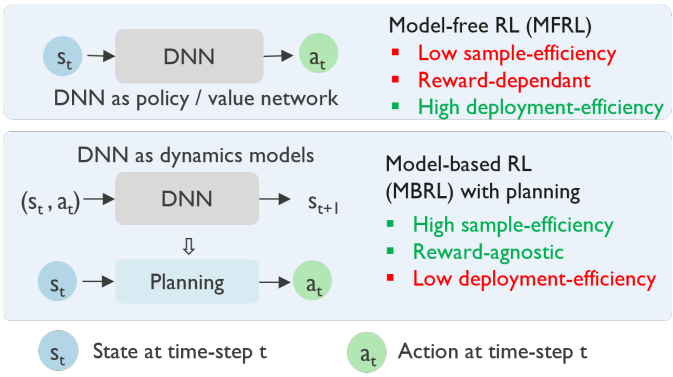


Figure 5.1: Conceptual role of DNNs for continuous control tasks and the general trade-offs among MBRL and MFRL.

planning process to still maintain desirable control quality, while increasing deployment efficiency.

Figure 5.2 depicts the general process of MBRL with planning, detailing the iterative process of DNN training and planning with DNN inferences. The planning process using these learned models involving multiple sequential inferences forms the major bottleneck for real-time deployment solutions [13]. Note that this chapter mainly deals with understanding the impacts of DNN block-quantizations for the planning process, whereas explorations on the quantized DNN training of the dynamics models themselves are discussed later in chapter 7. Additionally, the goal is not to claim that this is the best approach for continuous control, as there are multiple algorithms being developed continuously with different target optimizations. Our primary objective is instead to integrate these algorithms into a co-design framework alongside modern HW-efficient deployment solutions for DNNs. These platforms are crucial for advancing futuristic applications, catering to diverse requirements such as real-time control, algorithmic performance, and HW-efficiency across various domains.

This chapter is based on the accepted workshop article: Nitish Satya Murthy, Francky Catthoor, Marian Verhelst and Peter Vrancx. *"Quantized dynamics models for hardware-efficient control and planning in model-based RL."* In: Machine Learning and Principles and Practice of Knowledge Discovery in Databases. Ed. by R. Meo and F. Silvestri. Cham:Springer Nature Switzerland, 2025, pp. 196–209. [195].

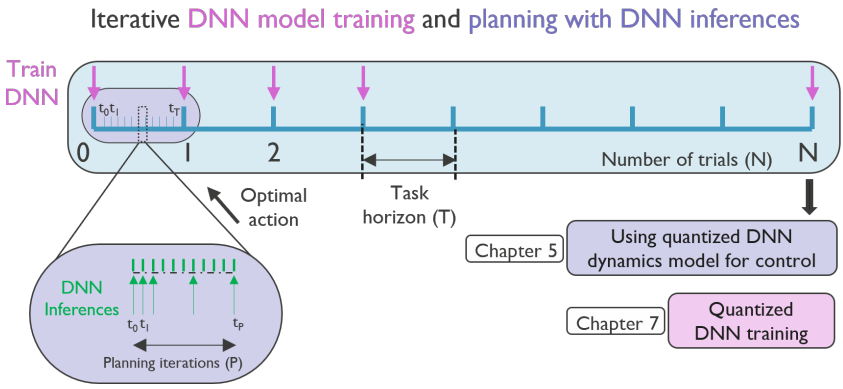


Figure 5.2: Timing details of MBRL algorithms with environment interactions involving N trials with each trial having T timesteps of task horizon. Each timestep within the task horizon requires multiple sequential DNN inferences to decide the optimal action.

5.1 Introduction

RL methods are one of the most promising algorithms for decision-making and control in various challenging environments [209]. Several variants of RL algorithms exist, each with different trade-offs and objectives [6]. For RL tasks, a major performance metric is sample-efficiency, i.e. how many interactions/samples are required for the agent to learn the task enabling effective control? This efficiency becomes crucial when solving complicated tasks in the real-world with real-time data. To overcome challenges in learning the task in the field, practitioners use model-based RL (MBRL) approaches that involve using a predictive model of the world (dynamics model). These methods are much more sample-efficient than their counterpart, model-free (MFRL) techniques, while reaching similar control performances [38, 95]. Figure 5.1 as shown earlier, conceptualized the two techniques - MBRL and MFRL, the role of DNNs, and the trade-offs among them. At a high level, the main objective for an agent of any RL algorithm is to decide which action (a_t) to perform at a time-step t in its current state (s_t) for maximizing the control performance often characterized by cumulative rewards across a defined task horizon. MBRL algorithms utilize a DNN-based dynamics model that tries to predict how an actual system might progress, by then selecting the actions leading to the most desirable control similar to a model predictive control (MPC) approach. In contrast, MFRL aims at directly outputting an action using a DNN by only observing the current state without any understanding of the world. Hence,

apart from the comparisons highlighted in Fig. 5.1, MBRL algorithms also tend to be more robust and safer in dynamic environments. The DNN training process in MFRL algorithms depend on the reward function, and hence need to be retrained when a new task or different reward function is desirable. While in MBRL algorithms, the training process is reward-agnostic, and therefore can be reused for customized tasks.

The most performant sample-efficient approaches make use of trajectory-sampling methods that generate a large number of possible future trajectories using the dynamics model [38]. This process is performed at every time-step in the real-world, whenever an action needs to be taken, based on the control frequency. Hence, while sample-efficiency and task-generalization improve with MBRL, the computational complexity for deployment becomes much higher. That is, planning and control using these dynamics models demands high compute power and long planning times [225]. The constraints become even harder with embedded robots that run on batteries and have limited hardware resources [148].

In this chapter, we focus on making the deployment of MBRL algorithms more hardware-efficient. While other works have focused on improving planning efficiency with fewer samples [173], our work optimizes the execution efficiency of predicting each sample trajectory. Specifically, we investigate whether the models can be optimized through quantization while maintaining control performances.

We utilize block floating point (BFP) datatypes due to their efficient fixed-point operations combined with high dynamic range through common exponents [45]. Quantization in MBRL algorithms for control tasks is under-explored and distinct from traditional DNN quantization, as the overall performance depends on a sequence of DNN inferences as opposed to a single inference. This means errors can accumulate and cause performance degradation. Additionally, the tasks involve continuous states/actions requiring regression-based DNN models, causing further challenges in quantizing the networks. Hence, we also account for flexible block precision and sizes, with different trade-off points to analyze the impacts on algorithmic performance.

To the best of our knowledge, this is the first work to explore quantized dynamics models for continuous control tasks. We moreover explore the effect of activation functions and their quantizations on the overall performance. We further justify our quantization choices through detailed analyses and empirical studies on standard continuous control tasks.

The following points highlight the main contributions in this chapter:

- Explorations of using efficient BFP MACs in the DNN dynamics model to boost MBRL deployment efficiency when compared to conventional FP32 models typically used for such tasks.
- Proposition and evaluation of different block quantization configurations to obtain the sensitivity of our optimized block choices (precision and size) towards control performance.
- Analyzing the performance trade-offs with detailed empirical studies on standard simulated MBRL control tasks.

5.2 Background theory on MBRL algorithm

In this section, we describe the background information for the concepts used in our work related to the specific algorithm of model-based RL with MPC for planning.

MBRL techniques are sample-efficient and also reward-agnostic since the model training mainly involves learning the state dynamics, after which, the task can be controlled according to any specified reward (a function of states). There were initial challenges for MBRL algorithms to reach MFRL performance, however, this was achieved by considering uncertainty in the dynamics model while making future predictions in the planning process, as done by [38] through probabilistic ensembles with trajectory sampling (PETS).

This uncertainty was induced through two main modifications: using probabilistic networks that predict distributions to account for noise in data, and using model ensembles to account for model bias. The algorithm of PETS along with the highlights of our block-quantization exploration is highlighted in Algorithm 2. The state trajectories predicted by these networks are as depicted in Fig. 5.3, which in turn provided estimates of cumulative rewards for a given action sequence. Cross-entropy method (CEM) [17] - a derivative-free optimizer was used to generate optimized input action sequences at different iterations, based on which the trajectories were ranked. The best performing action sequence after the last optimizer iteration was selected, and its first action was finally executed on the actual system.

The main takeaway for understanding the quantization impacts in this chapter is that the dynamics models under consideration are probabilistic ensembles that are combinations of multiple similar models, where each model predicts the mean and variance of the next state given a state-action input pair. Additionally, it is important to note that the model predictions degrade further

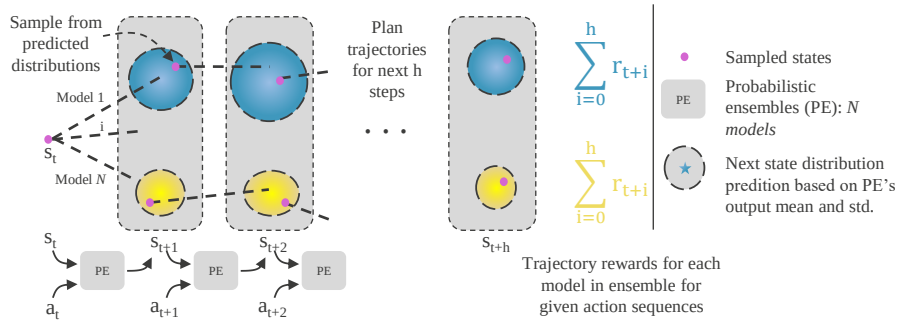


Figure 5.3: Methodology of trajectory sampling in PETS. With a state-action pair, the probabilistic ensemble network (PE) predicts the next state distributions. With initial state and an action sequence, the trajectories for each model in the ensemble are obtained by sequentially sampling states from the distributions. (Figure adapted from [38])

with quantization leading to potentially different optimal actions, and as a result different trajectories and control performances.

Algorithm 2 Model-based MPC algorithm – *PETS* (Used from [38]). Colored parts indicate where we perform block-quantization explorations.

Initialize data $\mathcal{D}$ with a random controller for one trial.
for Trial $k = 1$ to N **do**
  Train a DNN dynamics model f_θ given $\mathcal{D}$.
  (Quantized training explored in chapter 7)
  // Planning and control //
  for Time $t = 0$ to TaskHorizon **do**
    for iters $i = 1$ to CEMiters **do**
      Actions sampled $\mathbf{a}_{t:t+H} \sim \text{CEM}(\cdot)$
      Trajectory sampling of $\mathbf{s}_\tau^{(p)}$ using f_θ and $\mathbf{a}_{t:t+H}$ (Fig. 5.3)
      (Quantization explored here in chapter 5 at trial N)
      Evaluate actions as $\sum_{\tau=t}^{t+H} \frac{1}{P} \sum_{p=1}^{P} \text{Reward}(\mathbf{s}_\tau^{(p)}, \mathbf{a}_\tau)$
      Update CEM$(\cdot)$ distribution to get action sequences $\mathbf{a}_{t:t+H}$
    end for
    Execute only the first action $\mathbf{a}_t^*$ from optimal sequence.
    Record outcome: $\mathcal{D} \leftarrow \mathcal{D} \cup \{\mathbf{s}_t, \mathbf{a}_t^*, \mathbf{s}_{t+1}\}$.
  end for
  // Planning and control //
end for

5.3 Related work

Research on quantized DNNs for supervised DL has existed since long and is very extensive [65]. However, quantization in RL scenarios has received relatively lower attention. In model-free RL, quantization of policy and value networks have previously been explored. [106] looks at post-training quantization (PTQ) and quantization-aware-training (QAT) of DNNs that act as policy networks in different MFRL algorithms. [15] enables quantized training using 16-bit FP numbers of the DNNs in a SoTA MFRL algorithm – soft-actor critic [78] – that enables efficient control of the half-cheetah task. Authors of [73] aim at reducing the memory footprints for soft-actor critic through quantized state and action spaces. Hence, while some explorations exist for MFRL, quantization studies are even more scarce in MBRL. Additionally, the quantization error is more prominent in MBRL, because of compounding errors [95]. In MBRL, the model is used for ranking a trajectory's performance (sequence of inferences) and therefore compounds a single inference error across the complete trajectory (planning horizon). Hence, the quantization technique and the data format selection becomes even more crucial here so as to not incur performance degradation due to incorrect predictions.

Significant benefits have been demonstrated for BFP datatypes [43, 45] that provide high energy and area efficiencies for the MAC unit. These datatypes allow very efficient fixed-point multiplications, reducing the number of float computations for GeMMs, while still providing high dynamic range. Yet, the challenge exists in finding the optimal mantissa width and the block configurations. A small block size leads to higher hardware costs, while a large one leads to larger quantization errors. Hence, we systematically sweep across different combinations, observe the impacts on control performance and provide insights for the design choices to meet target requirements.

5.4 Block-quantized dynamics model for MBRL

In this section, we introduce a BFP quantized version of the probabilistic ensembles as the dynamics model for a hardware-efficient MBRL process. For a DNN GeMM between activations and weights, there exists a large search space with trade-offs between HW cost and algorithmic performance. We consider the following design choices to study the algorithmic impacts:

- Block shapes / Tiling strategy: Vector (V) and Square (S)
- Block sizes: 16, 32, 64, 128, 256

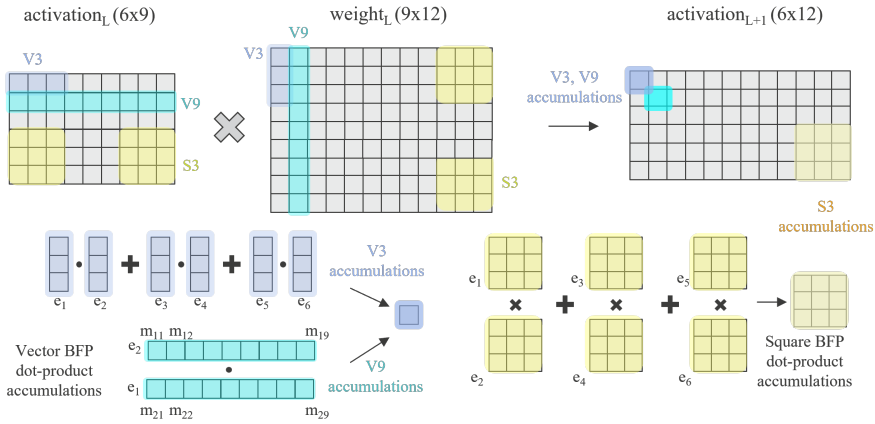


Figure 5.4: Illustration of block floating point based GeMMs with different quantization options. "V" and "S" denote vector and square blocks with a common 8-bit exponent respectively. The number in front of it represents the block size for efficient BFP dot-products. Each gray cell denotes a number in the tensor represented by x-bits for BFP-x configuration. Ex.: BFP8 (V9) represents vector blocks of length 9, with each no. in the block having an 8-bit mantissa.

- Block mantissa bits (weights and activations): 6, 8
- Block mantissa bits (inputs/outputs): 6, 8, 16
- Block exponent bits: 8
- Exponent selection strategy: Max. exponent in the block

Figure 5.4 illustrates the different block choices in a sample GeMM computation. Regardless of the original matrix dimensions, it is always split into complete blocks/tiles by appropriately increasing the size through zero-padding. For a given block size, note that both vector and square BFP GeMMs would have the same number of BFP dot products. It is essentially the mapping strategies which would change among them, parts of which are explored in our deployment studies in chapter 7. The square blocks would also be transpose-compatible with similar exponents within a block in the backward pass which becomes useful during fine-tuning or training process which we also exploit and demonstrate later in chapter 7. However, they would have higher quantization error as compared to vector blocks with finer exponent control. Block size and the mantissa bits for each number define the precision levels of the numbers within the block. Smaller blocks and larger mantissa bits offer higher precision, while design choices in the opposite direction increase the quantization error.

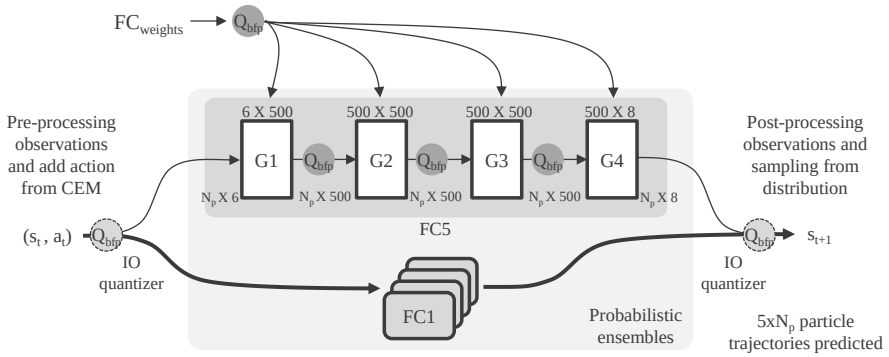


Figure 5.5: Probabilistic ensemble networks with simulated BFP GeMMs (Current network architecture shown for the cartpole task). Each network in the ensemble is a fully-connected network with inputs as current state-action pair and outputs a mean and variance predicting the distribution of next state. $N_p = 1600$, defines the number of trajectories to be predicted per model in the ensemble (400 different action sequences for 4 sampled particles).

Figure 5.5 illustrates the probabilistic ensemble network used in [38] and how our BFP quantizations are included in this work. Before each GeMM in the fully connected network, we simulate the BFP quantization using fake quantizer nodes parameterized with different choices mentioned previously in section 5.4. The states and actions in the environment, which are continuous in nature, are involved as the input and output of the networks. Hence, we allow the flexibility to have a different quantizer precision at the input/output of the DNN, which is crucial as we will see later in section 5.6. For all the quantization experiments, we used the dynamics model weights that was trained in FP32. We choose a Post training quantization (PTQ) methodology in our experiments to enable fast design space exploration.

5.5 Experimental setup

This section details the experimental setup for our work related to the continuous control tasks we select for benchmarking, the performance metrics we use for comparisons and the simulation setup to run the experiments.

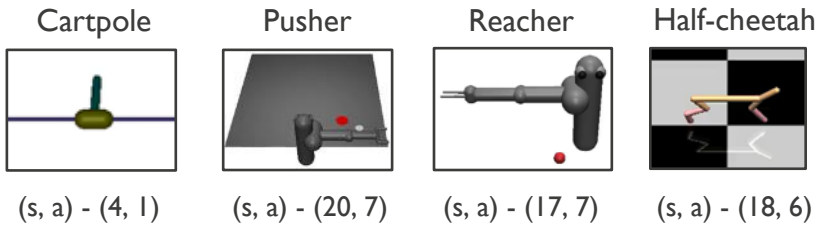


Figure 5.6: Continuous control tasks along with their state and action dimensions. [215]

5.5.1 Tasks

The target continuous control tasks are as shown in Fig. 5.6 are used for benchmarking the effect of quantized dynamics models in MBRL. The figure also indicates the state and action dimensions of the respective tasks, which are directly linked to the input/output sizes of the DNN model. MuJoCo [215] is used to simulate the system, which are used as the ground-truth dynamics of these tasks. Each task has different hyperparameter settings for the different components (planning parameters, CEM controller parameters, DNN dynamics model, etc.), which are used from the original implementation and are tabulated here in Table 5.1. Additional hyperparameter tuning for these networks is left for future work.

Task horizon defines the total number of time-steps the task is controlled and evaluated. Planning horizon denotes the number of time-steps that the model predicts in the future in the MPC process. The table also contains the network architectures used for different tasks. Activation choices explored include the original implementation's swish activation, along with a more hardware-efficient option of ReLU. The number of multiply-accumulates (MACs) required for a single inference of the model and the MACs for taking an optimized control action in the planning process are also reported in the table. The increased MACs ($\sim 100\times$) for action selection again highlights the MBRL computational complexity, as opposed to single-shot inferences.

5.5.2 Metrics of evaluation

Each observation state and action in the environment has an associated reward which is evaluated through the task's reward function. The total reward for

Table 5.1: Hyperparameter settings and no. of MACs for PETS dynamics models (FP32).

Hyperparameters		**Cartpole**	**Pusher**	**Reacher**	**Halfcheetah**
Network arch		FC3 (500)	FC3 (200)	FC3 (200)	FC4 (200)
Task horizon		200	150	150	1000
Planning horizon		25	25	25	30
CEM samples		400	500	400	500
CEM iterations		5	5	5	5
No. of MACs (GMACs)	1 inference	4.056	0.934	0.733	1.320
	1 action	507	116.750	91.600	198

a control task is the sum of all state and action rewards across its trajectory. Such an accumulated reward along the task horizon is used for evaluating the overall control performance. Similarly, a cumulative reward along the planning horizon is used by the CEM optimizer to evaluate the quality of different action sequences and output an optimal control action.

The rewards for the control tasks are functions of the state observations at any time and could be potentially determined through sensor readings on the system. As examples, the reward for the cartpole is a function of its observation states to balance itself vertically in an upright position and the reward for the reacher task is to move the tip of its 7 degrees of freedom robotic arm towards a goal.

5.5.3 Simulation setup

We run our experiments of training and evaluations on a single NVIDIA A100 GPU. We use QPyTorch [247], a quantization library with custom CUDA kernels for PyTorch to simulate BFP quantization. Note that the GeMM accumulations themselves are being done in floating point to be compatible with PyTorch operations, while we round the inputs and outputs of the GeMM to the specified BFP configurations. However, they support limited flexibility in block sizes (coarse), while we perform appropriate tiling according to our required block size before executing the quantization to allow flexible block configurations.

5.6 Results and analysis

We present the results of our experiments here, describing the performance impacts based on the major design parameters of the dynamics model: activation function, model quantization with different datatypes and block configurations.

5.6.1 Impacts of DNN activation function

We firstly evaluate the impact of activation functions, by using ReLU instead of swish for better HW efficiency. The performance numbers in Table 5.2 represent the mean and standard deviation of the accumulated rewards in the task horizon for 5 different runs with PETS FP32 dynamics models. It can be observed that ReLU networks perform similar to swish networks except for the more complicated task of half-cheetah. Hence, ReLU networks are chosen for the next experiments on quantized models for all tasks, while the last section would also include quantized swish network explorations specifically for the half-cheetah task.

5.6.2 Impacts of different quantization configurations for DNN dynamics model

We will see here how the control performance varies with different network quantizations including BFP as depicted in Fig. 5.7. We firstly evaluate the FP networks for different bit-widths and observe that other than for halfcheetah the FP8 networks perform quite well. However, FP units are computationally more demanding than fixed-point units, which can be efficiently utilized by BFP MAC units [45]. When we switch to a more HW-efficient BFP8 network with large blocks (single exponent for activation/weight tensors), we notice degradation in performance (red column bars). Hence, to reduce this loss, we can use smaller block sizes such as 16 to still be HW-efficient while improving

Table 5.2: Accumulated rewards for 5 random runs of each task (Tabular data shows: Mean ± std. across 5 runs). Higher rewards indicate better control performance.

Activation	Cartpole	Pusher	Reacher	Halfcheetah
Swish	179.48 ± 2.53	-80.07 ± 11.43	-43.47 ± 5.40	16063.20 ± 2820.51
ReLU	179.60 ± 1.53	-82.92 ± 9.08	-47.82 ± 7.63	6827.65 ± 228.29

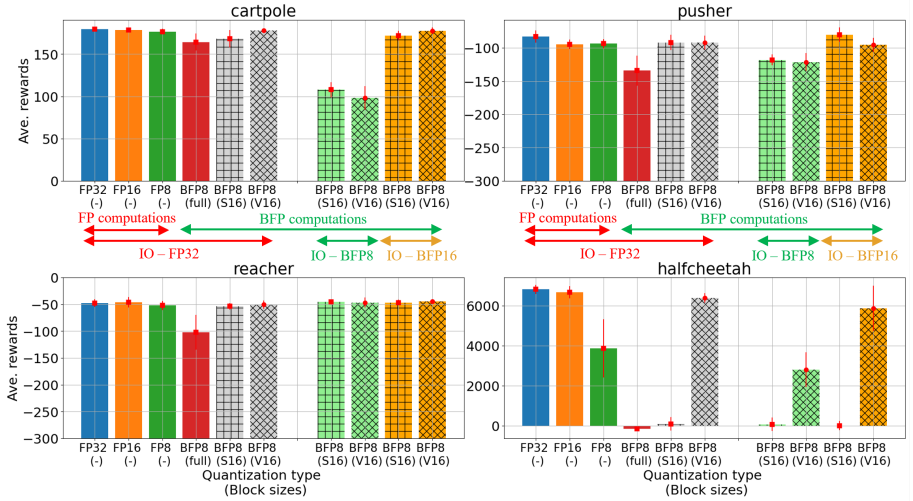


Figure 5.7: Impacts of different quantization datatypes and mixed-precision configurations of dynamics model on task returns. Bar plots show the mean and standard deviation of reward performance among 5 different runs.

algorithmic performance (gray hatched column bars - square and cross hatches represent square and vector blocks respectively). It is also interesting to see that even with these block sizes, the square configurations suffer severely in the case of halfcheetah, which we will analyze later in this section.

Note that the input/output, i.e. states and actions to the network were still kept in FP32 to isolate the network quantization effect against task-related quantizations. We also investigate if those could be optimized further. It is seen from the Fig. 5.7 that the networks do encounter performance loss with BFP8 inputs/outputs indicating a higher sensitivity requirement. However, better rewards could be reached by widening the BFP mantissas from 8 to 16 only for the inputs/outputs of the network. Hence, we still effectively utilize BFP computations for the quantized model while reaching performances close to FP32 models.

We now highlight the deployment advantages using BFP networks in Table 5.3. By using BFP8 networks and hence its higher MAC densities as compared to FP32, an estimated speed-up per area of up to ~ 8.8 times is expected [45]. The table also highlights the reduced memory footprints for MBRL process with BFP datatypes to store the model parameters, activation memory for DNN inference (batch size of 32), and that for the CEM optimizer to store different action sequences.

Comparison with conventional INT8 models

While INT8 models are one of the most well-studied and common quantization configurations, a major challenge in using standard INT8 models is calibrating the model well based on the benchmark dataset [145]. It is especially more challenging here since a sequence of inferences are used during planning and that they are highly dynamic, varying in real-time based on sampled states and actions, hence requiring different optimal scales for quantizations. In BFP quantization however, the scales are calculated on-the-fly at runtime based on the maximum element of the block during quantization, hence not having the calibration challenges that exist in conventional INT8 models with fixed scales.

For INT8 quantization experiments, per-tensor asymmetric and symmetric quantizations are used for activations and weights respectively for each model in the ensemble. Before evaluating the reward for the INT8 model, a dummy run is performed for calibration to obtain scales and zero-points. With these parameters, we simulate the INT8 quantization using fake quantizers in PyTorch and obtain the accumulated task-horizon rewards similar to our BFP evaluation setup. We tabulate these rewards alongside our BFP8 (V16) configuration for comparison in Table 5.4. The performance does increase with more calibration iterations, however, it is not feasible to perform such calibrations at runtime especially for real-world deployments. In spite of performing these calibrations, the rewards do not reach our BFP quantized model performance except for the reacher task, clearly suggesting the benefits of having flexible blocks with finer exponent control and dynamic runtime quantizations.

Table 5.3: Deployment and memory costs for PETS (FP32 vs BFP8).

Metrics		**Cartpole**		**Pusher**		**Reacher**		**Halfcheetah**	
Model precision		FP32	BFP8	FP32	BFP8	FP32	BFP8	FP32	BFP8
MAC density [45]		1x	8.8x	1x	8.8x	1x	8.8x	1x	8.8x
Memory (MB)	Model parameters	10.17 (1x)	2.71 (0.265x)	1.88 (1x)	0.5 (0.265x)	1.85 (1x)	0.49 (0.265x)	2.66 (1x)	0.71 (0.265x)
	Layer activations	0.242 (1x)	0.064 (0.265x)	0.107 (1x)	0.028 (0.265x)	0.105 (1x)	0.028 (0.265x)	0.138 (1x)	0.037 (0.265x)
	CEM optimizer	0.04 (1x)	0.03 (0.75x)	0.350 (1x)	0.263 (0.75x)	0.280 (1x)	0.210 (0.75x)	0.360 (1x)	0.2650 (0.75x)

Table 5.4: INT8 model rewards comparison.

Tasks	BFP8 (V16) IO-BFP16	INT8, IO-FP32 (task calibration iters.)		
		0	5	100
Cartpole	177.5 ± 3.8	0.8 ± 0.6	135.9 ± 20.7	156.1 ± 15.0
Pusher	-95.2 ± 10.4	-157.4 ± 4.2	-143.6 ± 10.6	-144.5 ± 11.9
Reacher	-44.8 ± 4.4	-174.2 ± 25.5	-54.6 ± 17.8	-53.8 ± 9.3
Halfcheetah	5882.9 ± 1134.5	-399.0 ± 241.0	-95.1 ± 183.9	-55.9 ± 198.5

5.6.3 Impacts of mantissa bit-width and block sizes

Having larger block sizes is desirable if the HW supports it, as the common exponent along the block amortizes the dot product costs through efficient fixed-point multiplications. On the other hand, it can also induce larger quantization errors and might lead to inefficient computations for small matrix multiplications due to lower utilization. From Fig. 5.8, we firstly see that it is again desirable to go for vector-based blocks instead of square to obtain higher rewards. From the results, we see that there is only a slight degradation of performance with increasing sizes. However, reducing the mantissa bit-width from 8-bits to 6-bits has a higher impact on the control performance. Hence, the quantization configurations need to be carefully selected based on the target task, while some tasks such as reacher are more robust to quantizations as compared to other tasks such as halfcheetah.

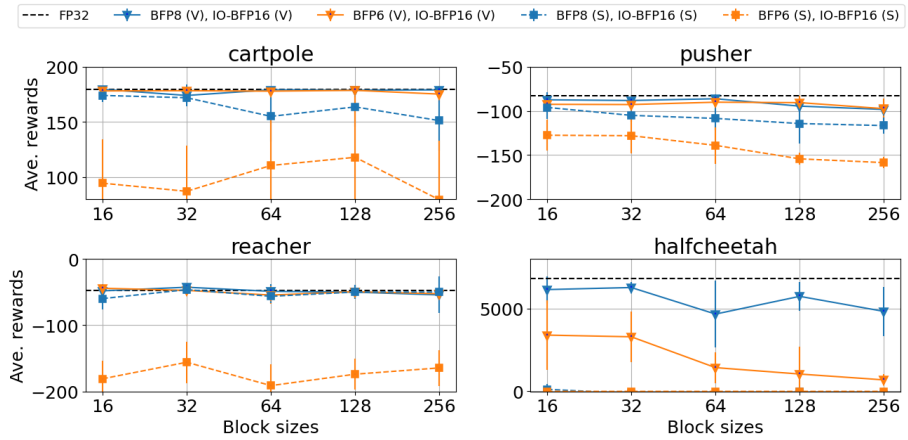
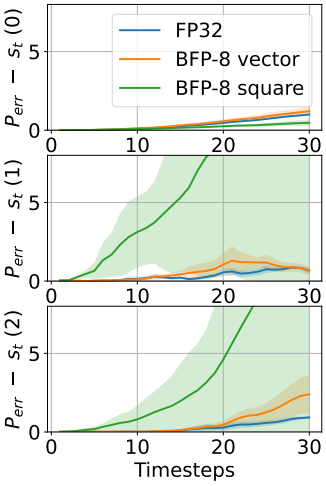
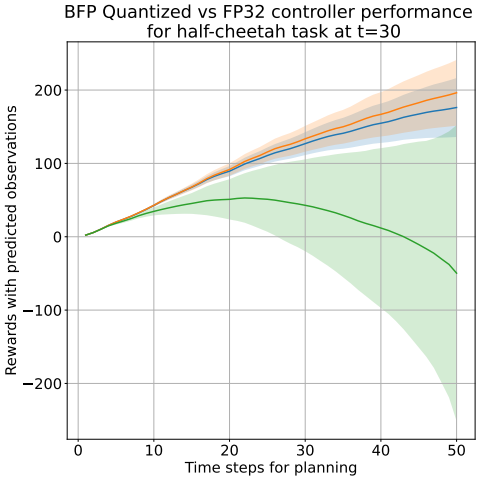


Figure 5.8: Impacts of BFP mantissa bit-widths, block shapes and sizes.



(a) Prediction errors with ground-truth model (normalized to peak FP32 model error).



(b) Rewards for predicted states at each time step.

Figure 5.9: Quantized model controller predictions for half-cheetah task.

5.6.4 Quantization analysis for halfcheetah tasks

Analyzing the square vs vector blocks in halfcheetah task

It is to be noted again that even though the BFP dot-product lengths are the same for square and vector blocks, the number of elements sharing the same exponent is larger for square compared to vector blocks, hence leading to a higher quantization error. Figure 5.9 highlights the reason for worse performance using square blocks for halfcheetah. The prediction MSE errors with respect to ground-truth dynamics are seen in Fig. 5.9a for the first 3 out of 18 states of the half-cheetah. It clearly shows that square block quantized model's predictions deviate away from vector's or FP32 model predictions and worsens as we predict further into the future. Its impact on reward predictions is also seen in Fig. 5.9b, demonstrating why vector block controller decisions would be more similar in action-selection to FP32 controller and hence provide better task control.

Quantized swish network rewards for halfcheetah task

We saw previously that the swish networks reached much better FP32

Table 5.5: Accumulated rewards using quantized swish networks for the half-cheetah task.

Activation function	FP32	BFP8 (S16)	BFP8 (V16)
ReLU	6827.65 ± 228.29	122.20 ± 323.15	5882.91 ± 1134.57
Swish	16063.20 ± 2820.51	1184.65 ± 483.57	9416.95 ± 3428.97

control performance for halfcheetah task than ReLU networks, hence probably demanding more non-linearity. While ReLU activations are very HW-efficient to implement, swish activations enable modelling more complex non-linear functions. This increased capacity benefits the halfcheetah task since it is a relatively more complex and sensitive task compared to others. It is also sometimes known to converge on control behaviour that leads to a rolling motion to move faster for achieving higher rewards. While this is a unique solution when compared to a running motion, it can lead to numerically unstable solutions [244].

Nevertheless, we also examine the impacts on quantizing such swish networks that deliver higher control performance. Table 5.5 shows the reward values for FP32 vs BFP8 networks with swish and ReLU activations. As can be seen, the quantized swish networks perform better than ReLU with similar configurations, but has a higher variance among different runs. It is also observed that swish networks undergo higher performance degradation than ReLU networks indicating a lower robustness to quantization. Hence, there exists a trade-off choice that needs to be studied based on the control task requirements for selecting the activation function.

5.7 Conclusion

In this chapter, we shifted our focus from conventional supervised learning tasks towards a different modality of tasks known as continuous control tasks. Continuous control tasks require actions to be taken for a system in an environment to obtain desirable control, where our aim, in particular, is to achieve more data-efficient and HW-efficient controllers.

With this objective in mind, we demonstrated a block floating point (BFP) quantized dynamics model that reduces the computational complexity of the planning process in a popular data-efficient model-based RL technique – PETS [38]. The HW-efficient fixed-point multiplications in BFP MACs, makes the MBRL process more feasible towards real-time deployment. We also explore several BFP design choices for the dynamics models and investigate different

quantizations of the task states and actions to understand the impacts on MBRL control performance of standard continuous control tasks. In addition to enabling the use of faster and denser BFP MACs, our approach significantly reduces the memory requirements by up to $\sim 3.78\times$ compared to FP32 networks, all while maintaining comparable control performance.

While both chapters 4 and 5 explored BFP datatype explorations for data-efficient workload explorations across different task modalities, however, they were mostly explored at an algorithmic level to analyze the impacts of quantization. We did not discuss any HW metrics such as instruction or cycle counts to run the workloads with block-scaled datatypes like BFP. In the next chapter, we also include investigations of HW deployments with these datatypes while also comparing with standard datatypes such as FP32 and INT8 formats to provide a more holistic view.

Chapter 6

Block-scaled GeMM deployments using general-purpose vector instructions

The previous chapters 4 and 5 mainly involved algorithmic explorations of optimized quantization configurations for challenging data-efficient scenarios in diverse modalities such as supervised learning and continuous-control tasks intended for deployment on resource-constrained devices. However, the explorations were purely a block-quantized numerical sensitivity analysis, without specific HW deployment studies. In this chapter, we address the research question defined in section 1.2: *Q2. How do we efficiently deploy DNNs with block-scaled formats using general-purpose vector instructions?*

Prior works have discussed the benefits of block-scaled formats in terms of having tightly co-optimized HW-SW solutions, however, the HW platforms considered were mainly ASIC designs as a custom accelerator [45][43][60]. While these platforms do provide high efficiencies, they are limited in their programmability, have high non-recurring engineering costs, and are often not widely available for quick explorations. In contrast, vector instruction-set based processors, also referred to as vector processors here for simplicity, overcome the aforementioned challenges. Designed for handling vector data, they present an attractive choice for effective data-parallel workload computations while ensuring high programmability and the potential for high energy-efficiencies through parallel-

computing compared to scalar processing [171]. However, the deployment of flexible block-scaled kernels on variable length vector processors have not yet been investigated. With this target, we explore scenarios of deploying block-scaled kernels on ARM-SVE based processors by developing our own intrinsics-based GeMM kernel for high vectorization. We propose different configurations of this kernel that could be suitable in diverse scenarios based on varying user needs such as feasible pre-calibration efforts, different accuracy-throughput trade-off targets, etc. Additionally, the programming model is vector-length agnostic (VLA), i.e. the same kernel code can be deployed on HW platforms with different vector lengths, providing huge portability gains. We additionally provide an inter-block accumulation pipeline in INT32 instead of FP32 which is commonly used for BFP formats, hence referring to our deployment dataformat as block-scaled integer formats (BS-INT) instead of BFP. These explorations also provided insights on new instructions that might be useful to increase code performance, and/or increase deployment options for block-scaled kernels potentially through new vector instruction-set extensions.

This chapter is based on the published article: Nitish Satya Murthy, Francky Catthoor, and Marian Verhelst. *"Optimization of block-scaled integer GeMMs for efficient DNN deployment on scalable in-order vector processors."* Journal of Systems Architecture 154 (2024): 103236 [194].

6.1 Introduction

Instruction-programmable processors are well-suited for flexible workloads, while conventionally achieving much lower performance than custom accelerators [29, 155] for massively parallel tasks such as deep learning. However, with the growing popularity of DNNs, there have also been rising efforts for programmable processor design towards accelerating AI workloads. Vector processors that enable SIMD parallelism are a leading choice for accelerating DNN workloads, as the computations are majorly comprised of simple MAC instructions, which can then be applied throughout the hardware (HW) vector length for multiple data elements. Recent trends in processor design for AI accelerations are towards long vector architectures which not only reduce memory accesses but also enable further exploitation of massive DNN data parallelism, increasing energy-efficiencies while maintaining compatibility for diverse domains [76, 172].

These efficient vector processing architectures also retain their flexibility by being instruction-programmable, with several vector/SIMD instruction sets (ISAs) being available. Some of the popular ones such as ARM and RISC-V each have their own vector Instruction Set Architecture (ISA) extensions -

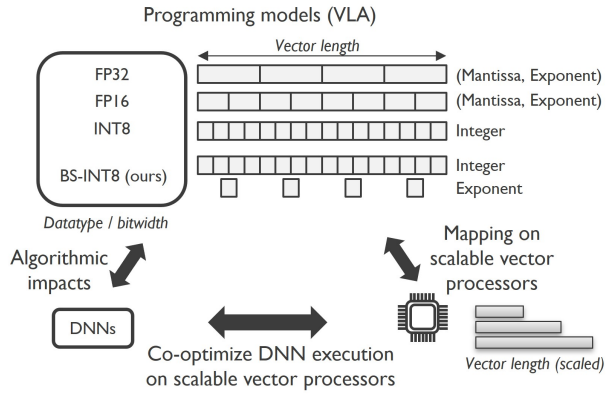
ARM scalable vector extension (SVE) and RISC-V vector extensions (RVV) respectively, which are also vector length scalable leading to VLA programming models, to enable portability across different HW vector lengths [167, 204]. In this work, we mainly explore the widely-used ARM SVE architectures which are compatible with scalable vector units ranging from 128 to 2048 bits. SVE also includes different degrees of parallelism in their integer MAC instructions – simple scalar-vector product (*axpy*), 4-way dot-products (*dp*) and also matrix-multiply accumulations (*mm*) [203]. While SVE was created for high performance computing (HPC) targets and server workloads, recent developments also focus on edge architectures such as ARM Neoverse [170]. In such diverse applicability, VLA greatly reduces the SW development efforts and enables faster co-design explorations. The HW-implemented maximum vector length is the vendor's choice, with examples of SVE-enabled hardware like A64FX (512 bits) [136], Neoverse N2 (128 bits) [169], Neoverse V1 (256 bits) [197], etc., while their programming model remains the same due to binary compatibility.

With quantization of DNN parameters, we can further increase the throughput for the same HW vector lengths through subword parallelism. However, the model accuracy reduces with quantization and the specific impacts are workload-dependent, hence demanding effective co-optimizations. Programmable processor optimization is moving towards efficient longer vector architectures [136, 167], while datatype exploration is focusing on the quantization impact analysis for algorithmic performance through block-scaling [47]. Our goal in this chapter is to find co-optimized solutions while not requiring custom HW to retain flexibility and widespread applicability. Block-scaled custom processors contain specific HW blocks for performing several involved computations efficiently like accumulation alignments, block normalization, and requantization. In contrast, we aim to achieve this on a vector processor flexibly. As shown in Fig. 6.1a, the explorations of mapping trending DNN quantization methodologies on upcoming vector-scaled hardware are essential for optimized system-level designs, providing further insights for co-optimization solutions to meet target specifications.

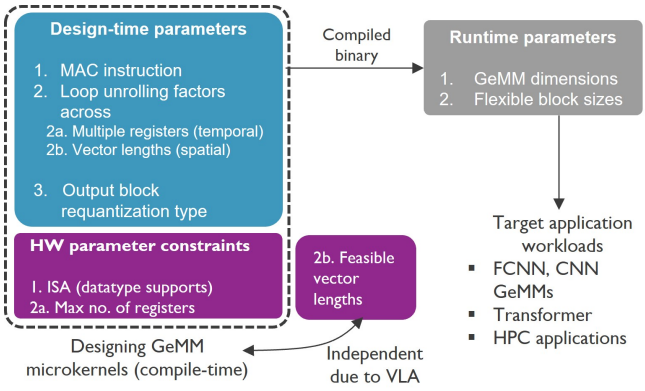
In this chapter, we focus on optimizations of deploying our block-scaled integer GeMMs (BS-INT8) on scalable vector processors using ARM-SVE based VLA programming models as depicted in Fig. 6.1b and analyzing the different trade-offs through extensive design space explorations.

Our main contributions are as follows:

- Design of optimized VLA programming models for block-scaled integer GeMMs (BS-INT8) on fully-programmable scalable vector processors. (Sections 6.3.1 and 6.3.2)



(a)



(b)

Figure 6.1: (a) Co-optimizations of deploying VLA programming models of DNNs on scalable vector processors. (b) Description of our VLA programming model for deployment of BS-INT8 GeMMs.

- Mapping exploration, design, and analysis using different microkernels for throughput optimization. (Section 6.3.1)
  - Introducing flexible block sizes and granularities, which can provide up to 2× higher speedups for 2D over conventional 1D blocks for GeMM computations in standard ResNet20 architectures. (Section 6.3.3)
- Establishing and analyzing end-to-end computation and fused-requantization operations without utilizing floating point vector units. (Section 6.3.4)
- Profiling cycle counts for our different BS-INT8 GeMM configurations on ResNet20/56 architecture on CIFAR10/100 datasets and baseline comparison to FP32 and INT8 datatypes (Section 6.5.6)

This chapter is organized as follows: Section 6.2 details the background and related work. Section 6.3 describes our methodology for optimizations and the corresponding design parameters for the block-scaled kernels. We detail the benchmarking tools, profiling techniques and general experimental setup in Section 6.4. Section 6.5 presents and analyzes the obtained results. Lastly, we conclude our work in Section. 6.6.

6.2 Background and related work

Vector processors Vector processors are very useful for workloads with massive data parallelism such as DNNs. Traditional vector units offer several advantages such as energy-efficiency and increased throughput. However, the ISA developed for one HW vector length was not generally applicable to a different length architecture [204]. This led to the development of scalable vector architectures like ARM SVE, where the ISA is agnostic to the HW vector lengths [204]. ARM SVE is an extension introduced in v8-A architectures for vector-length agnostic (VLA) programming models, in contrast to its previous ARM NEON architecture (128-bit fixed-length SIMD units). It reduces the programmer's effort to map kernels on different vector length hardware through predication, which also automatically handles tail-end loops. Figure 6.2 shows the principle of predication enabling fine control of vector instruction impacts, which together with a VLA programming model enables diverse deployments. Depending on the datatype, workload size and the HW vector length, each bit in the predicate register decides the activity of its respective vector register lane. It allows for quicker DSEs, more efficient and portable code with only a single implementation - either relying on auto-vectorization of compilers, handwritten intrinsics/assembly code, automated code generation [213], etc. This implementation choice depends on specific use-cases and involves a clear

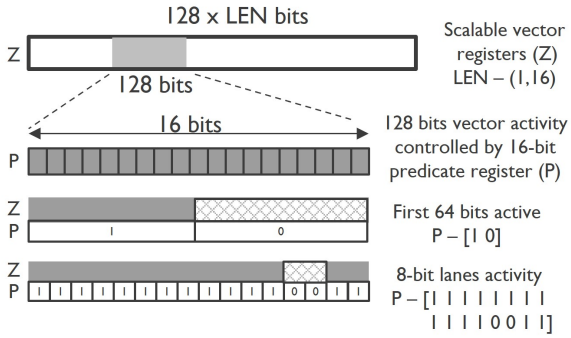


Figure 6.2: ARM SVE includes 32 vector registers of scalable length 128xLEN bits (Z) and 16 predicate registers of length 16*LEN bits (P) (1 bit for the smallest controllable granularity of 8-bit lanes in a vector register). [204]

trade-off between ease of programmability and effectiveness. VLA models also enable binary compatibility with different vector lengths. Due to these advantages, modern CPUs are moving towards scalable vector processors with widespread applications in domains ranging from the cloud to the edge. Research has also progressed on different kernel deployment analyses on ARM SVE HW with targets such as HPC workloads [183], large integer GeMMs [52], FP32 CNN deployments [76], posit emulated convolutions [40], etc. Some of the ARM SVE instructions used for our block-scaled kernel design are summarily shown in Table 6.1. In the rest of the section, we introduce DNN quantization (more details in section 3.2), briefly discuss the prior block-scaled techniques and its processor implementations in literature. Lastly, we also explore and distinguish our approach with other optimized GeMM deployment algorithms.

DNN quantization Quantization in DNNs usually involve its parameter representations in floating point (FP) and fixed point number formats (FXP/INT). FP numbers contain an exponent and a mantissa part, while FXP numbers only have a mantissa part composed of fractional and integer bits. Due to the separate exponents, FP numbers generally have a higher dynamic range than FXP numbers. However, this comes at the cost of more demanding FP deployments, due to the alignments of exponents during every accumulation step. Extensive research on number representations has been done to analyze the deployment trade-offs for quantized DNNs as depicted in Fig. 6.3, including but not limited to varying bit-widths for different FP and FXP parts.

Block-scaled numbers, also called as microscaling formats (MX) [47], involve sharing a scale or exponent with a group of numbers (block size), while the

Instruction	Description
`svld1`	Loading data into vector registers.
`svld1rq`	Loading 128-bits data replicated along the total vector length.
`svld1ub`	Extended load to 32-bit vector registers.
`svdup`	Duplication of a scalar value along total vector length.
`svst1b`	Storing truncated 8-bit values into memory.
`svdot_lane`	Indexed 4-way INT8 dot product accumulating in 32-bit integer outputs, with lane index specifying which 8-bit integer group is selected for first vector operand.
`svasr / svlsl`	Right / Left shift lanes of vector register by shift values in another vector register operand.
`svmaxv`	Return maximum value of the vector register.

Table 6.1: ARM Vector Instructions used for the implementation of block-scaled GeMM kernels.

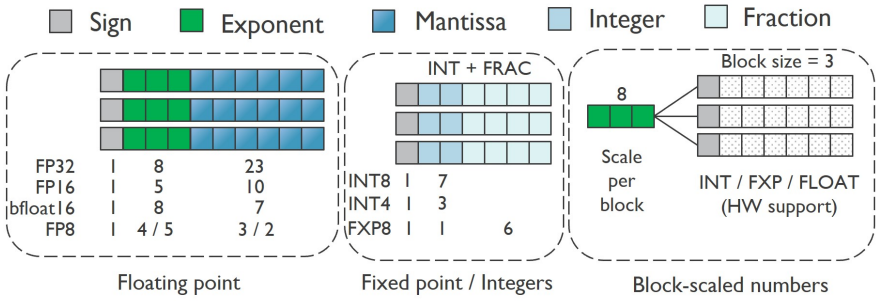


Figure 6.3: Common number representations in conventional and block-scaled formats.

numbers themselves can be represented in low precision FP or FXP formats [47]. The specific block-scaled representation choice is usually dependent on several factors - HW constraints such as computational support, memory, power, latency, etc., and workload demands such as accuracy, flexibility, etc. We refer to the version of FXP/INT block operands as BS-INT throughout this chapter. This greatly helps in compressing the data to reduce memory footprints, data transfer costs and also helps in reducing the number of accumulation alignments when compared to FP formats.

Block-scaled DNN processors Block floating point (BFP) representations are the most popular block-scaled formats, targeted initially towards digital signal processing applications [98]. Such block-scaled formats are shown to yield huge benefits in DNN computations, with recent propositions to make them a standard datatype [47]. Table 6.2 summarizes the key differences in various

Table 6.2: Block-scale based computing processors for DNNs.

Processor type	Reference processors (Details)	Block sizes / shapes (Runtime)	Flexible vector lengths?	MAC instructions	Accumulation type	Explored requantizations types	Operand datatypes (Design-time)
Custom accelerator	HBFP [51]	Fixed	-	Custom	FP32	Dynamic	INT 8/12
	MSFP [45]	Fixed	-	Custom	FP32	Dynamic	INT 4/6/8
	VSQUANT [43] (2-level scaling)	Fixed	-	Custom	INT (custom width)	Dynamic	INT 3/4/6/8
	Block minifloat [60] (Kulisch accumulator)	Fixed	-	Custom	INT (custom width)	Dynamic	FP 5/6/7/8
	DBPS [118]	Flexible	-	Custom	FP32	Dynamic	INT 4/8/16
Programmable vector processor	INTEL AVX-512 [48]	(tensor-level)	x (512 bits)	Dot products	INT32 / FP32	Dynamic	INT16
	ARM NEON [20]	(tensor-level)	x (128 bits)	Axpy	INT16/32	Static	INT 8/16
	Ours (ARM SVE)	Flexible	✓ (128 - 2048 bits)	Axpy / Dot product (dp) / Matrix-multiply accumulate (mmla)	INT 8/32	Dynamic / Static	INT8 Baselines (non-blocked): INT8, FP32

block-scaled processor implementations. The majority of related processors are of custom accelerator types [43, 45, 153, 185]. Computations of each block are done with FXP accumulations while partial product accumulations across blocks are mostly performed in FP32 [45, 51, 185]. Additionally, [51] also stores the intermediate activations in expensive FP32 formats to retain compatibility with non-MAC based DNN operations to avoid accuracy loss, while [47] optimizes this further by utilizing bfloat16 storage formats. It has previously been shown for vector processors that the removal of FP unit led to ∼2x smaller and more energy-efficient computing units [7], providing a potential optimization opportunity that we utilize in our work. Authors in [20, 48] have investigated the deployment of block-scaled DNNs on programmable processors, however, with fixed-length vector units. [48] also consider 16-bit operands with accumulations in FP32 for overflow management. Furthermore, they only consider tensor-level exponents in their exploration, while we employ flexible block sizes and shapes, and also consider different target HW vector lengths for wider design space exploration. Lastly, we also investigate different block requantization strategies through layer fusion for more efficient deployment [4].

Optimized GeMM deployments Since GeMMs are one of the most crucial linear algebra operations with widespread applicability, they have been studied in-depth for efficient deployments resulting in multiple standardized libraries [16, 234]. BLIS [217] is one such library developed for high-performance and portability. It involves partitioning large matrices into smaller sub-matrices for effectively streaming data through the cache hierarchy. While other libraries also accomplished this, BLIS also attains easy usage and development by performing the GeMM with 5 loops written in high-level C language, around a small architecture-specific assembly-based microkernel as seen in Algorithm 3. It has also inspired several similar optimized GeMM deployment explorations for different processors, with varying target requirements [2, 10, 179]. While we also utilize blocks in GeMM computations similar to BLIS, our motivation for using blocks stems from a block-scaling datatype perspective that benefits low-precision DNN deployments [45]. This leads to certain key differences in both implementations as illustrated in Algorithm 3 and tabulated in Table 6.3.

Firstly, block-scaling enables us to use 8-bit operands instead of typical FP32/FP64 used in BLIS. These block-scaled datatypes, with INT8 operands and a shared block exponent, do need more architecture-specific intrinsics code than BLIS for handling accumulations and requantizations. However, it does enable us to run GeMMs effectively on a vector processor through subword parallelism providing efficient HW-SW co-optimization capabilities for DNN deployments. Such optimizations are also not easily generated by standard compilers, making these logic blocks crucial for reaching good performances. Further details regarding these logic blocks will be discussed

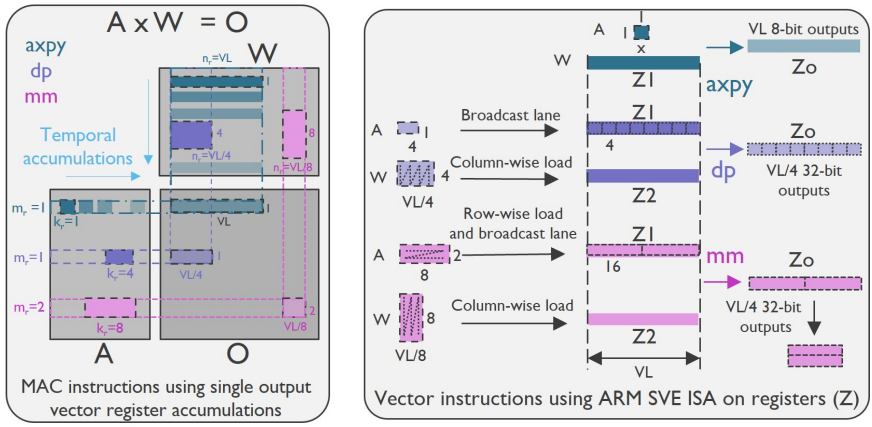
Algorithm 3 BLIS algorithm and high-level differences to our implementation (marked)

```
1: for j_c = 0; j_c < n − 1; j_c+ = n_c // Loop 1 do
2:   for p_c = 0; p_c < k − 1; p_c+ = k_c // Loop 2 do
3:     B(p_c : p_c + k_c − 1, j_c : j_c + n_c − 1) → B_c
4:     for i_c = 0; i_c < m − 1; i_c+ = m_c // Loop 3 do
5:       A(i_c : i_c + m_c − 1, p_c : p_c + k_c − 1) → A_c
         // (Macro-kernel)
6:       for j_r = 0; j_r < n_c − 1; j_r+ = n_r // Loop 4 do
7:         for i_r = 0; i_r < m_c − 1; i_r+ = m_r // Loop 5 do
8:           // Loop 6 (Micro-kernel)
9:           for p_r = 0; p_r < k_c − 1; p_r+ = 1 (4/8) do
10:            C_c(i_r : i_r + m_r − 1, j_r : j_r + n_r − 1)
11:              += A_c(i_r : i_r + m_r − 1, p_r) · B_c(p_r, j_r : j_r + n_r − 1)
12:            // Exponent alignment logic (intrinsics code)
13:          end for
14:          // Fused requantization logic (intrinsics code)
15:        end for
16:      end for
17:    end for
18:  end for
19: end for
```

Table 6.3: Conceptual and implementation differences of our block-scaled GeMMs with BLIS algorithm.

Parameter	BLIS GeMMs	BS-INT GeMMs (Ours)
Complexity / Portability	Only microkernel in assembly	C language vector intrinsics
Datatypes / bitwidths	Typically high-precision (Float-32/64 bits)	Low-precision (Block-scaled formats - 8 bits)
Microkernel accumulation steps	Typically 1 (outer product)	1 / 4 / 8 with INT8 subword parallelism
Blocking stage	Before packing routines, to form macrokernels	After packing routines, partitions microkernel accumulation
Block size	Typically $\gg$100, selected based on cache sizes	8-128 with DNN accuracy consideration

later in Sections 6.3.2 and 6.3.4. In addition to computational benefits, with similar problem sizes, block-scaled datatypes have a lower memory footprint reducing the stress on cache hierarchies. With moderately-sized matrices which is common in DNN applications, we directly use the macrokernel without any blocking in loops 1, 2 and 3 in Algorithm 3.



(a) **(b)**

Figure 6.4: (a) GeMM (A x W = O) computation examples using different ARM SVE MAC instructions spatially unrolled along VL. (All dimensions are in bytes unless specified otherwise). (b) Mapping onto vector registers–Z, using ARM SVE vector instructions.

6.3 Methods

The previous section briefly discussed the works done on block-scaled representations. However, to the best of our knowledge, this is the first time that an optimization study of mapping flexible block-scaled kernels on vector processors is performed using VLA-based programming models. Specifically, in our block-scaled representation (BS-INT8), we look at flexible blocks with each number being an 8-bit signed integer, along with a common 8-bit unsigned block exponent. We will mainly investigate the most computationally intensive DNN kernels–GeMMs as depicted in Fig. 6.4a, prevalent in workloads discussed previously in chapters 4 and 5. To also ensure flexibility and generalized applicability, we perform this exploration without any custom HW unit or instruction, targeting commodity SVE-supported hardware. This exploration is done through design and optimization of the VLA programming model by exploiting the following techniques and their respective *design parameters*:

- Loop unrolling on vector processors for low-precision GeMMs through ARM SVE MAC vector instructions across - vector lengths (*spatial factor*), multiple output registers (*temporal factor*)
- Block-scaled integer GeMMs for HW-SW co-optimizations - *Block size, B*

- Flexibility of block size and shapes through programmability - *Block shape (1D or 2D)*
- Requantization of GeMM outputs back to block-scaled formats - *Static / Dynamic*

Applications can be run effectively through appropriate selections at compile/runtime as was described earlier in Sec. 6.1, Fig. 6.1b. Optimum configurations depend on specific workloads, and can be tuned following a generic design methodology which we will introduce at the end of the section. The rest of the section explores more in detail for each of the aforementioned optimization techniques.

6.3.1 Loop unrolling on vector processors

Vector processors have multiple vector registers, each with lengths able to store multiple elements, providing huge parallelization opportunities. A large influence is based on the GeMM microkernel design, which is usually a small, highly-optimized (architecture-specific) computational kernel typically written in hand-coded assembly or intrinsics. Deployment studies are typically performed through multi-level loops around this microkernel. We investigate loop unrolling across each vector register length (VL) spatially as illustrated in Fig. 6.4 for GeMM kernel accumulations using VLA ARM SVE programming models. We consider deployments on single-core processors in this work, while in multi-core processors, spatial unrolling across these output registers is another parallelization option. Since ARM SVE compilers do not yet support instantiating an array of registers, investigating flexible microkernel sizes can be done effectively through automatic code generation [2]. We restrict our experiments to a specific microkernel size, instead focusing on comparing different datatypes and leave further microkernel shape optimizations for future work. However, the same techniques are applicable for other microkernel shapes as well.

The specific unrolling factors (m_r, n_r, k_r), similar to the microkernel shapes in BLIS considered for a GeMM are tabulated in Table 6.4, which are also dependent on the specific ARM SVE MAC instruction. The 3 integer MAC instructions considered are: scalar-vector product (*axpy*), lane-based 4-way dot product (*dp*), lane-based matrix-multiply and accumulate (*mm*). Lane-based implies that one part/lane of a vector register can act as a broadcasted input operand for all other lanes of another input operand. This increases the arithmetic intensity as a single vector load enables multiple compute combinations through lane selections. An illustration of single output vector register accumulation for all considered MAC instructions is shown in Fig. 6.4a.

Table 6.4: MAC instructions with the microkernel accumulation steps and dimension. VL represents the specified vector length in bytes.

MAC instruction	Inputs / Output bit-width	Accumulation steps	Microkernel dimensions	
		k_r	m_r	n_r
Scalar-vector product (axpy)	8 / 8	1	8	VL
Lane-based 4-way INT8 dot product (dp) (1x4) x (4x1)	8 / 32	4	8	VL/4
Lane-based matrix-multiply accumulate (mm) (2x8) x (8x2)	8 / 32	8	8	VL/8

Additionally, Fig. 6.4b depicts how these GeMM inputs and outputs are mapped onto vector registers for different MAC instructions. Note that, in contrast to BLIS, the input and output bit-width may vary, and accumulation step size is not necessarily 1. The accumulation is also performed across multiple vector registers, with temporal accumulations in an output stationary fashion, reducing memory accesses and reusing registers effectively. These different GeMM configurations utilizing varied MAC instructions are efficiently designed using suitable vector instructions to enable high performance.

6.3.2 Block-scaled integer GeMMs

Figure 6.5a illustrates our block-scaled (BS-INT8) GeMM setup with input activation/weight integer and exponent parts. Note that the sizes of exponent matrices are smaller than their integer counter-part, defined quantitatively by the design parameter, block size (B). A dot product among two block-scaled vectors involves fixed-point accumulations of integer parts and a simple addition of the exponent parts [45]. The previous subsection covered the parallelization of integer GeMMs on vector processors, with microkernel accumulation taking place temporally in an output stationary fashion. However, in a BS-INT8 GeMM process, during the temporal accumulation, alignments of the accumulations are required at block boundaries due to having different exponents as depicted in Fig. 6.5b. At each block boundary, the partial product conversion and inter-block accumulations in FP32 is possible, or a soft-float type of accumulation can be performed. We utilize the soft-float accumulation with corresponding integer and exponent parts, which emulates an approximated floating point addition (more details in Appendix. B). Algorithm 4 provides more implementation details about this alignment for a single accumulation register in BS-INT8

Algorithm 4 Code logic with ARM SVE instructions for block-scaled alignment during BS-INT8 GeMM computation.

Vector registers: Vreg; Exp. matrices: `Aexp`, `Wexp`; Integer matrices (after packing): `Aint`, `Wint`; Block index: `b`; Indices along GeMM accumulation: i_r, j_r, p_r

1: Computations of intra-block INT32 accumulations with INT8 operands after loading left/right matrix operands into registers
   $\mathbf{R_A} \leftarrow$ `svld1rq(Aint[`i_r`][`p_r`])`
   $\mathbf{R_W} \leftarrow$ `svld1(Wint[`p_r`][`j_r`])`
   $\mathbf{Acc_0} \leftarrow$ `svdot_lane(`R_A`, `R_W`, index=[0])`

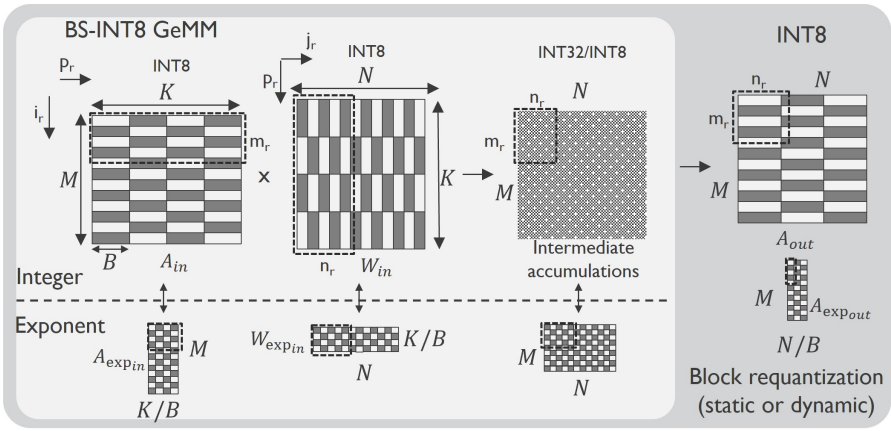
2: Load 32-bit extended 8-bit exponents of a vector of **Wexp**
   $\mathbf{Wexp_{new}} \leftarrow$ `svld1ub_u32(pred, &Wexp[b+1][`j_r`])`

3: Add scalar exponent of A to a vector exponent **Wexp**
   $\mathbf{Oexp_{new}} \leftarrow$ `svadd_n_u32(pred, Wexp`$_{new}$`, &Aexp[`i_r`][b+1])`

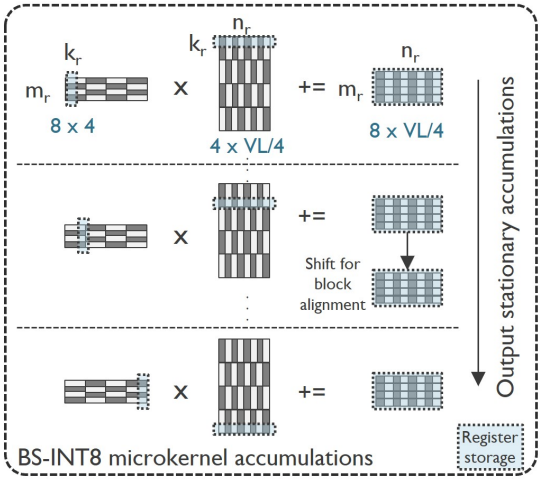
4: Compute absolute shifts between old and new output exponents
   $\mathbf{E_{sh}} \leftarrow$ `svabd_u32`(pred, Oexp_{new}, Oexp_{old})

5: Shift accumulation register by vector of shifts
   $\mathbf{Acc_0} \leftarrow$ `svasr_s32(pred`$_{right}$`, Acc`$_0$`, E`$_{sh}$`)`
   $\mathbf{Acc_0} \leftarrow$ `svlsl_s32(pred`$_{left}$`, Acc`$_0$`, E`$_{sh}$`)`

GeMM process. Assuming 32-bit accumulation bit-widths being calculated for each block with INT8 operands, we also need to get the exponents to similar bit-width to enable feasible vector operations. The process involves loading weight exponents across multiple column values into a vector register; add corresponding scalar activation exponent to get new output exponent vectors; subtract previous stored block output exponents to obtain shift values that are finally applied on the accumulation vector register. This is also performed for every output accumulation register, and throughout the GeMM accumulation length. Although this creates some computational overhead, the alignment only needs to be performed at boundaries defined by block sizes. Hence, tuning the block size not only impacts quantization errors as discussed in previous chapters, but also on mapping performance leading to an important trade-off potential.



(a) BS-INT8 GeMM computations with specific unrolling factors.



(b) Temporal microkernel accumulations in an output stationary format across blocks.

Figure 6.5: (a) BS-INT8 GeMM with separated integer and exponent parts and requantization for next layer at the register level without much data transfers and include in the microkernel. (b) Illustration of microkernel accumulations in BS-INT8 format with alignments at block boundaries (Block size - B).

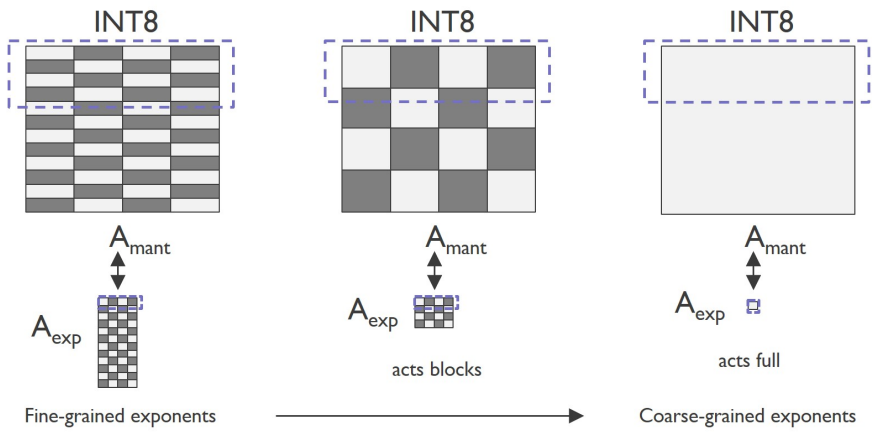


Figure 6.6: Flexible granularity of activation blocks in the computation of BS-INT8 GeMM. Dotted lines represent microkernel accumulation dimensions.

6.3.3 Flexible block granularities - 1D vector vs 2D blocks

Since block sizes have a significant impact in BS-INT8 co-optimizations, making further explorations of how blocks are formed, are crucial for achieving better performance. We observed previously that the exponents needed to be handled as scalars for activation (left) matrices, also in accordance with traditional vectorized GeMMs for integer parts. This leads to a potentially more optimal design decision for sharing exponents across activation rows in addition to columns. Fig. 6.6 illustrates possible block-scaled quantizations for activation blocks with varying granularities. The finer blocks involve a 1D vector-level granularity, which had some computational overhead. Alternately, at the other direction lies the most coarse level, to have a single scale for the complete matrix. While coarser blocks reduce computational effort required in block alignments, the quantization error increases due to higher compression ratios. There are additional issues with requantizations when blocks exceed microkernel shapes which is analyzed in the next section. Hence, an optimized block shape should include as many rows as possible that fit within the microkernel, and large enough to reduce scalar exponent overheads, implying a 2D activation block shape with similar rows as the microkernel a good co-optimized block choice.

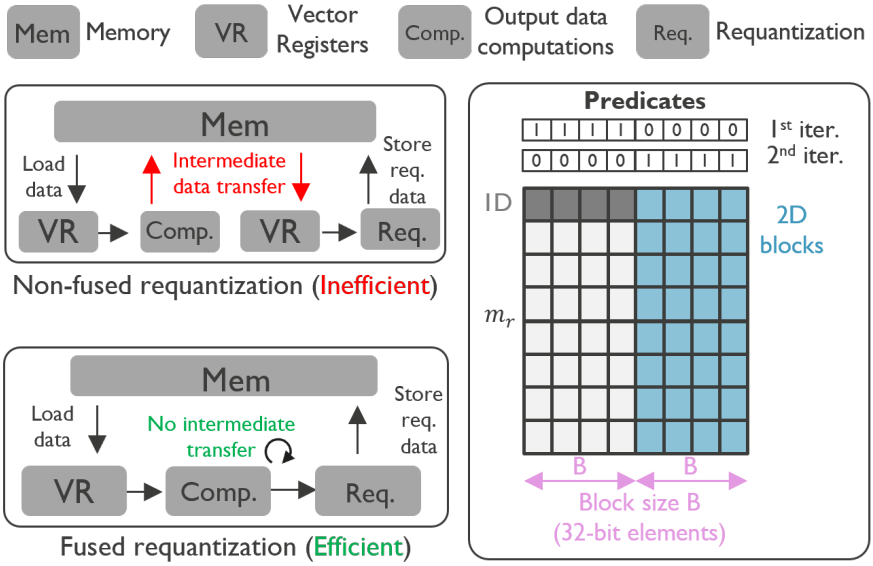


Figure 6.7: Illustration of fused and dynamic requantization in vector processors through block-size based predication. Optimal scenario is when block size fits perfectly in the microkernel shapes ($m_r \times n_r$), if not, we perform multiple loops with predication selecting the specific block elements.

6.3.4 Requantization methodologies - static vs dynamic

The requantization of accumulated BS-INT8 GeMM outputs to block format as also previously depicted in Fig. 6.5, is necessary if these outputs are to be effectively used as inputs for other downstream GeMMs. A possible solution is to store intermediate outputs in non-block formats such as FP32 / bfloat16 [47, 185], and perform the quantization operation later, which does provide an advantage of performing any other operations which might require non-compressed higher-precision inputs [51]. However, this not only involves utilizing expensive FP vector units, but also prevents us from performing efficient fused requantization. Instead, we focus on output requantization at the register level in a fused-operator manner to store them at lower precision in memory, reducing expensive data transfer costs as depicted in Fig. 6.7. Additionally, there are different ways to do this requantization - static and dynamic.

Static requantization requires output block scales to be provided as inputs, that are calculated beforehand through calibration, increasing the pre-deployment efforts [57]. However, this option does enable highly flexible block selections.

It also further eases the requantization process significantly since there are no overheads in runtime calculations.

```
Vaccs[0:7] // After microkernel accumulations in macrokernel
for (iters = 0; z < n_r/B; z += 1):
  pred = initialize predicates to restrict instructions to blocks
  if 1D blocks:
    Ecurr = svld1ub_u32(pred, exp_ptr) // Get current exponents
    Vcls = svcls_s32(pred, Vacc0) // Get leading sign bits
    Vnorm = svsub_u32_z(pred, Ecurr, Vcls) // Normalize exponents
    Emax = svmaxv_u32(pred, Vnorm) // Maximum block element
    Emax = 24 - Emax // Shift value for INT32 to INT8 truncation
    Vshifts = svsub_u32(pred, Emax, Vshifts) // Diffs with max
    Vacc0 = svasr_s32(pred, Vacc0, Vshifts)
    svst1b_s32(pred, ptr_out, Vacc0)
    ...
    // Repeat for all output accumulation vector registers (Vacc
        [1:7]) in microkernel
  else if 2D blocks:
    Ecurr = svld1ub_u32(pred, exp_ptr) // Get current exponents
    Vcls0 = svcls_s32(pred, Vacc0) // Get leading sign bits
    Vcls1 = svcls_s32(pred, Vacc1)
    Vnorm = svmin_s32(pred, Vcls0, Vcls1)
    Vcls2 = svcls_s32(pred, Vacc2)
    Vnorm = svmin_s32(pred, Vcls2, Vnorm)
    ... // Repeat for all vector registers (Vacc[1:7])

    Vcls7 = svcls_s32(pred, Vacc7)
    Vnorm = svmin_s32(pred, Vcls7, Vnorm)
    Vshifts = svsub_s32(pred, Ecurr, Vnorm)
    Emax = svmaxv_u32(pred, Vshifts) // Maximum block element
    Emax = 24 - Emax // Shift value for INT32 to INT8 truncation
    Vshifts = svsub_u32(pred, Emax, Vshifts) // Diffs with max
    Vacc0 = svasr_s32(pred, Vacc0, Vshifts)
    svst1b_s32(pred, ptr_out, Vacc0)
    // Repeat for all output accumulation vector registers (Vacc
        [1:7]) in microkernel
```

Code 6.1: Pseudo code for performing dynamic requantization for block-scaled integers to normalize with maximum elements within a block and store truncated INT8 outputs. The main difference between 1D and 2D blocks relates to whether the maximum element of the block is only related to that vector register or if the maximum across different vector registers needs to be considered for normalization.

On the other hand, dynamic requantization is performed by normalizing the outputs to the maximum exponent within each block at runtime. This leads to more robust quantizations for highly dynamic data in the field [195], however, at the cost of increased complexity. Figure 6.7 and the pseudocode 6.1 illustrate a predicated dynamic requantization process to convert the microkernel outputs

of a GeMM to BS-INT8 format. Note that we focus on performing this in a fused-operator manner, wherein the data is not required to be transferred to and from the memory, rather it is converted at the register storage level. The non-fused versions would require intermediate storage outside registers, hence demanding a large data transfer and memory access overhead. Although, in fused versions we can only consider the limited configurations in which the microkernel sizes are larger than (integer multiples) or equal to the block sizes. For example, to perform for fused dynamic requantization with a microkernel size of 8×8 output accumulations, the maximum block size for 1D block in each row is 8 with other feasible values being 2 / 4, while greater block sizes would require storing intermediate results in memory. As shown in Fig. 6.7, we implement this on vector processors by selecting specific elements corresponding to the block partitions along vector lengths through predicates in multiple iterations. Although suboptimal, it enables more flexibility than relying on a single scenario of when the blocks perfectly fit in the microkernel width. Among the active elements, the maximum output is identified, and all other numbers within the block are normalized to this maximum and truncated to 8-bits thereafter (more details in Appendix. B). It should be noted that only the operations which necessitate a block level comparison - finding maximum elements, is performed redundantly in multiple iterations. The final output matrix would have each block element as 8-bit integers, with the maximum scale of block forming the common exponent. This methodology is also applicable for 1D and 2D blocks due to high programmability, with changes mainly in finding maximum value within the blocks in a single vector register or instead across multiple vector registers in the microkernel.

With such a large design space that exists for various configurations, we need to study and analyze them so as to obtain optimal choices based on user specifications. Figure 6.8 depicts a general design guideline that helps user selections for kernel design choices (compile/runtime). Dynamic requantization is generally preferred over static, when deployment ease is a priority or when the data is very dynamic requiring runtime calibration. Thereafter, the microkernel can be designed by selecting the specific MAC instruction to use and the unrolling parameters, while satisfying the target HW constraints. Finally, the block sizes can be selected based on a trade-off analysis for the specific workloads and target optimizations.

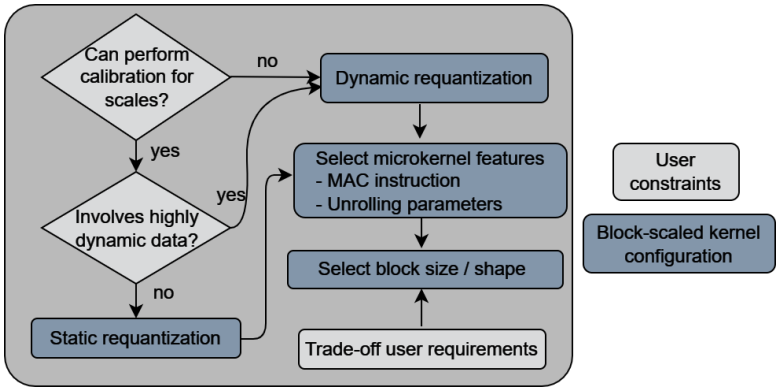


Figure 6.8: Design guideline for selecting kernel parameters based on workload requirements.

6.4 Experimental setup

Benchmarking and profiling tools: We use the following open source tools to benchmark our kernels:

- armie [5] - ARM instruction emulator (Version 22.0) to get SVE emulated instruction counts. We use markers to get instruction counts in the region of interest to isolate the kernel computation. Counts are obtained for emulated SVE and total instructions, providing an insight into the vectorization achieved in the kernel. While this tool can be used natively on any ARM v8 architecture irrespective of SVE-support, it does not yet support the matrix-multiply (mm) instruction.
- gem5 [14] - Gem5 is used to get the instruction / cycle count measure for the binaries in system evaluation (SE) mode. Gem5 public version currently also supports the INT8 mm instruction. Cycle counts provides a better system level comparison metric than instruction counts. The simulation setup we used in all our experiments are as shown in Table 6.5.

Code implementation details: The kernel codes are written in C with intrinsics support for SVE instructions through ARM C Language Extensions (ACLE). Enabling similar performance explorations through compiler auto-vectorization is left for future work, which might enable more portability and easier kernel design . All matrices are stored in row-major order.

Table 6.5: Gem5 simulation setup details.

Parameter	Description
Core Type	in-order MinorCPU
Core Frequency	1 GHz
ISA	ARM with SVE
L1 Cache	64 kB
L2 Cache	1 MB
Cache line size	64 B
DRAM Size	8 GB
Mode	Syscall Emulation
gem5 Version	v23.0

Compiler and host platform details: ARM clang version 22.1 for armie runs on aarch64 platforms and ARM gcc 12.3 cross-compiler for gem5 runs were used on an x86_64 host. A static compilation, with 'O3' optimization level and armv8.6-a+sve2 target processor architecture are used throughout all our experiments.

Target benchmarking kernels: The following list includes the kernels that we plan to profile to explore the impacts of our kernel designs.

- Baseline datatype comparisons - FP32, INT8 GeMMs for different matrix sizes.
- Baseline standard library - BLIS FP32 GeMMs (ARM-SVE) for similar matrix configurations through blocked GeMMs with cache hierarchy consideration.
- BS-INT8 GeMM computation complexity excluding the output requantization step to get insights into optimal kernels for static requantization scenarios along with different activation block granularities.
- Dynamic requantization for BS-INT8 - Specific blocksize and VL combinations to study the overheads of fused dynamic requantization.

6.5 Results and analysis

In this section, we discuss the results and analysis of our experiments. The specific questions we investigate are:

- How do the instruction counts and cycle counts scale with increasing vector lengths?(Sections 6.5.1 and 6.5.2)
- How do the overheads of performing block-scaled GeMMs vary with different combinations of block shapes and vector lengths? How does the performance trend for GeMMs with varied matrix sizes? (Sections 6.5.2, 6.5.3 and 6.5.4)
- With the inclusion of fused requantizations, how do the BS-INT8 GeMMs perform with dynamic and static configurations, compared to baseline datatypes? (Section 6.5.5)
- How do different BS-INT8 kernels perform for GeMM deployments in a ResNet20 CNN architecture [82] for CIFAR10/100 datasets [108]? (Section 6.5.6)

Notations: Note that we denote all results with different datatypes explicitly. The INT8 and BS-INT8 microkernel terminologies used here are as reported in Table 6.4. For example, *dp* (INT8) and *dp* (BS-INT8) denote microkernels with dot product instructions. Each MAC instruction is associated with only 1 microkernel shape and hence not specified everywhere.

6.5.1 Baseline GeMM performances

As an initial step, we set up baselines for our BS-INT8 GeMM explorations by profiling GeMMs with FP32 and INT8 datatypes. The integer MAC operations would be similar to the ones described in Table 6.4, while for FP32 formats we use lane-based fused-multiply and add operations (FMLA). We also explore the scaling impact of vector lengths and the code implementations of naive vs. optimized intrinsics.

An initial comparison is shown in Table 6.6, we observe close to linear scaling of speedups with increasing vector lengths in terms of instruction counts. We compare the intrinsics-based code total (T) instructions to a naive (N) 3 for-loop GeMM implementation for *dp* (INT8) and observe that the intrinsics code is $>23\times$ faster due to better vector instruction utilization. It also tabulates the percentage of the total *dp* (INT8) computations used for packing weight and activation matrices. It is observed that the packing instruction overheads are very less in comparison to the total instruction count ($\sim$4%), while providing huge benefits in terms of contiguous memory accesses. The comparison of the *dp* (INT8) and FP32 GeMMs results in *dp* (INT8) GeMMs being around $3.6\times$ faster (lower instruction counts). The vectorization percentages for both datatypes with intrinsics-based implementations were found to be $\sim$75%, indicating good vectorization across increasing vector lengths for a high-parallelism GeMM

Table 6.6: Instruction count splits and vectorization percentages for dp (INT8) and FP32 GeMM kernels of (128x128) x (128x128) matrix sizes.

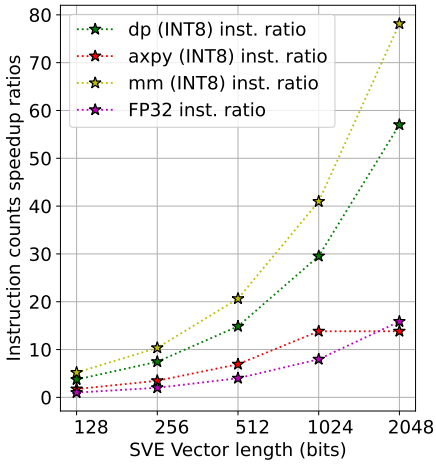
SVE VL (bits)	Naive INT8 (N) (N/T)	Intrinsics code (INT8)			Intrinsics code (FP32) Total (FP32_T) (FP32/INT8)	INT8 SVE vec. (%)	FP32 SVE vec. (%)
		Total (T)	Acts pack (%)	Weights pack (%)			
128	6292660 (23x)	273562	3.2	2.01	1033713 (3.59x)	75.0	73.16
256	3278004 (23x)	136858	3.3	2.02	517105 (3.59x)	74.9	73.13
512	1770676 (25x)	68506	3.5	2.04	258801 (3.58x)	74.6	73.07
1024	1017012 (29x)	34330	3.9	2.07	129649 (3.56x)	74.2	72.96
2048	1017012 (58x)	17242	6.3	3.00	65073 (3.45x)	73.0	72.72

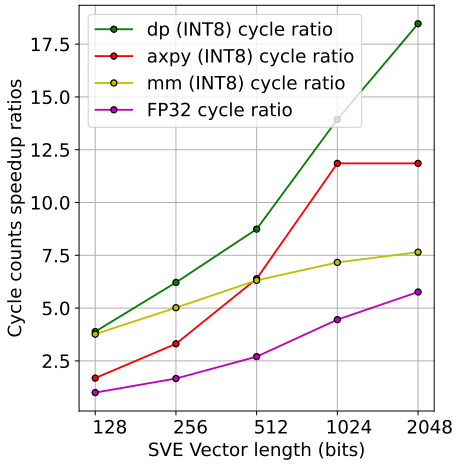
workload. Additionally, the instruction counts from both armie and gem5 setups were found to be very similar for the intrinsics code implementation despite using different compilers.

6.5.2 BS-INT8 GeMM kernel performance comparisons

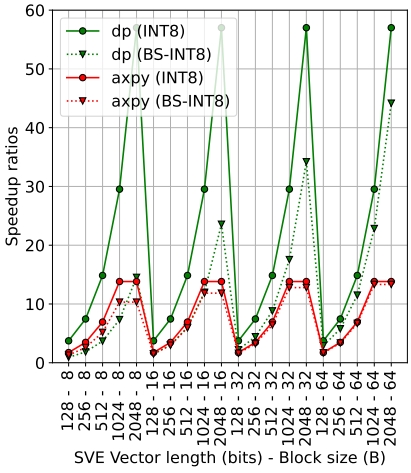
In this section, we compare the instruction and cycle counts for different INT8 and BS-INT8 microkernels implemented in intrinsics to analyze system-level block scaling overheads and also the trends with increasing vector lengths. Figure 6.9a highlights the INT8 kernel comparisons with different MAC instructions (*axpy*, *dp*, *mm*). Note that in the y-axis, a normalization is performed to an FP32 instruction or cycle count with 128-bits SVE configuration to capture the combined impacts of vector length scaling and varied datatypes. We firstly point out that scatter-store instructions are utilized to correctly align the output stores for the *mm* kernels, while for others contiguous stores are used. The impact of this is clearly visible in the figure, wherein the *mm* MAC kernels outperform the dot product MAC kernels in instruction counts, whereas they perform worse in cycle count comparisons due to the system-level overheads of non-contiguous stores.

Figs. 6.9b and 6.9c compares the GeMM performances for different combinations of vector lengths (bits) and block size. Note that non-blocked INT8 instructions are independent of block sizes. We observe in Figs. 6.9b and 6.9c that *dp* clearly outperforms *axpy*, due to having good parallelism in its MAC instruction and effectively covering the square matrix dimensions better with increasing vector lengths. To understand the previous point, we observed a saturation in instruction counts with increasing lengths for *axpy* deployments due to the fact

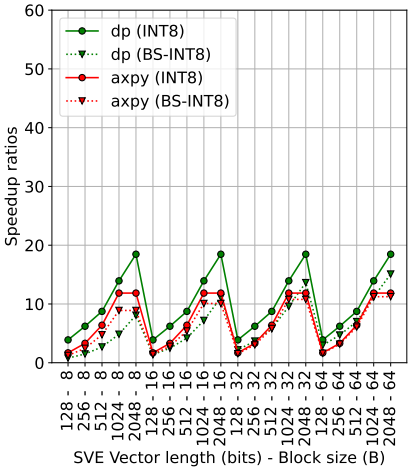




(a) MAC instruction comparisons – *axpy*, *dp*, *mm* (inst / cycle ratios)



(b) Instruction count ratios



(c) Gem5 cycle count ratios

Figure 6.9: (a) Relative speedups of instruction counts and cycle counts with different MAC instructions, showcases the difficulty with *mm* kernels requiring scatter store operation, while instruction counts scale very well, cycle counts end up as worse as FP32 especially with rising vector lengths. System-level comparisons of gem5 cycle count ratios are significantly lower than instruction count ratios especially for increasing vector lengths. Relative speedups of (b) instruction counts and (c) cycle counts for BS-INT8 and INT8 GeMMs using different microkernels. Notice that sometimes there is saturation, as the vector registers unrolling exceed the matrix boundaries. The block size is only relevant for BS-INT8 formats, while the speedups remain the same for others. All normalizations are with FP32 SVE-128 bit deployments.

that they exceed the matrix sizes and are hence made partially inactive through predication leading to low utilization. Hence, we select the *dp* kernels for further evaluations due to having better and more reliable system-level performance. In the later sections, unless otherwise specified, the terminologies INT8 and BS-INT8 would refer to a *dp* MAC instruction microkernel.

While VLA models help in binary compatibility, it is not always assured to provide high utilization as it depends on several factors such as problem sizes, code implementations, compiler reliability and mapping procedure. Another point to note is that, for the same vector lengths, increasing block sizes leads to faster execution as the alignment overhead reduces. This alignment overhead is also noticed across each combination by comparing the INT8 kernels with the BS-INT8 version, although with the benefit of increased dynamic ranges exhibited by block-scaling. Furthermore, among the two figures, although the trend remains similar, instruction counts overestimated speedups compared to cycle counts for increasing vector lengths, as system-level measures with also cache hierarchies are not considered.

6.5.3 Optimizing BS-INT8 GeMMs through activation block granularity

Plotting *dp* kernels in isolation with the baseline kernels in Fig. 6.10, we observe that *dp* (BS-INT8) outperforms most FP32 and performs worse than

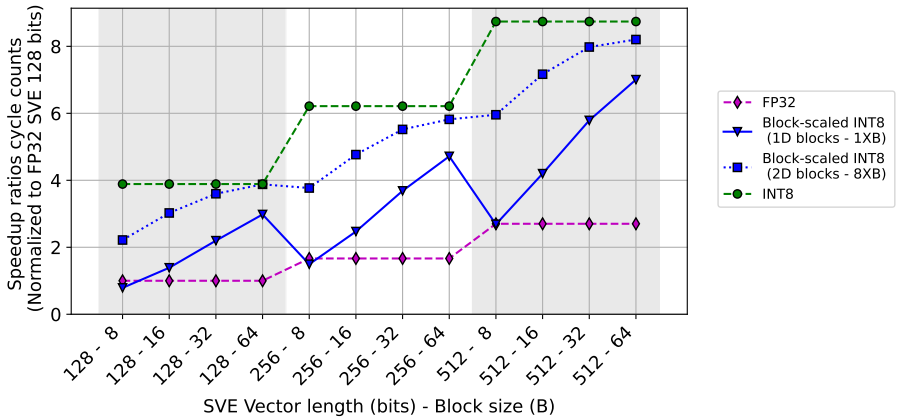


Figure 6.10: Relative speedups of cycle counts for BS-INT8 (*dp*) vs FP32 and INT8 (*dp*) with different granularities of activation blocks (1D and 2D).

all *dp* (INT8) versions as expected due to the alignment and exponent handling overheads. While we do observe improvements compared to FP32, the overheads are still relatively high for fine-grained 1D blocks. With the reasoning in our design decision for activation block granularity explained in Sec. 6.3.3, we do indeed observe in Fig. 6.10, that we can recover the loss in performance and get closer to INT8 performance with coarser (m_rxB) 2D blocks.

6.5.4 Analyzing GeMM performances for different matrix dimensions

While earlier experiments investigated a fixed-size GeMM kernel, the impacts of scaled matrix sizes is studied here. We analyze this by isolating individual matrix dimensions during the scaling while keeping the other two dimensions fixed, in addition to configurations with scaling all dimensions. Note that we do not yet perform BLIS-like blocking here, the whole GeMM is considered at the macrokernel level with potential further optimizations by utilizing such a blocking strategy. Figure 6.11 depicts the scaling effects clearly, where different matrix dimensions and datatypes are profiled. Since the problem size varies, normalization would be required to study the trends which is done here by evaluating the amount of MACs computed per clock cycle defining the computational efficiency of the kernel. We firstly confirm that the previous insights of baseline comparisons to FP32 and INT8 with different BS-INT8 configurations through block shape and size are still observed here: 2D blocks outperform 1D blocks, and increasing block sizes leads to increasing speedups/computational efficiency. We also observe a decreasing trend of computational efficiency with increasing dimensions, indicating a potential improvement opportunity through efficient cache blocking. However, in spite of this performance degradation the speedups are observed to be more than FP32 BLIS standard library baselines.

6.5.5 Static and dynamic requantization strategies for BS-INT8 GeMMs

While our previous comparisons were mainly for GeMM computations, the effects of requantization were still not analyzed. With static quantization, the overhead would be negligible as the scales to shift are calibrated beforehand, whereas they need to be computed at runtime for dynamic quantizations. An important distinction here is that very large block sizes are suitable for larger throughputs in static requantization, while they are sub-optimal for dynamic case as it prevents us from performing fused operations and requires expensive

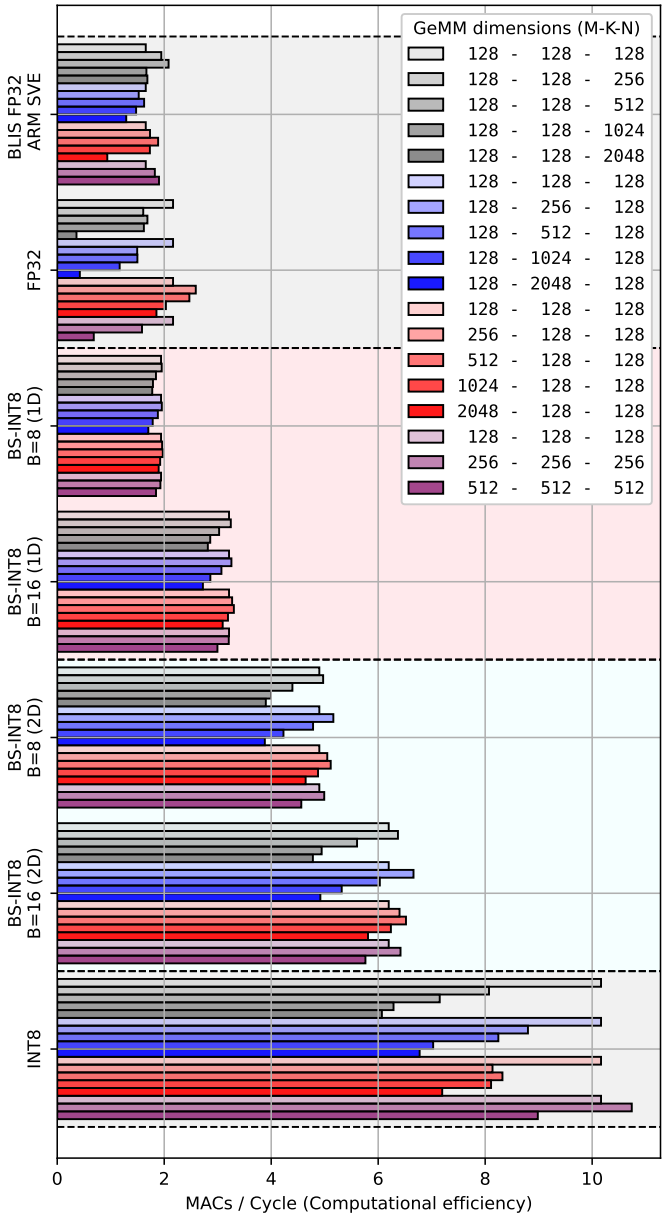


Figure 6.11: Impacts of scaling along different GeMM dimensions evaluated on a 256-bit SVE vector length configuration. Plotting performance metric of MACs per cycles demonstrates the computational efficiency. Trend shows higher computational efficiency for fewer GeMM MACs or sizes, with reducing performance along either increased dimensions. The previous trends established for different blocksizes and shapes for BS-INT8 is also seen here for different scaled GeMMs. (Background colours representation: Gray colour – baselines; pink and cyan colour – Our BS-INT8 kernels with 1D and 2D blocks resp.)

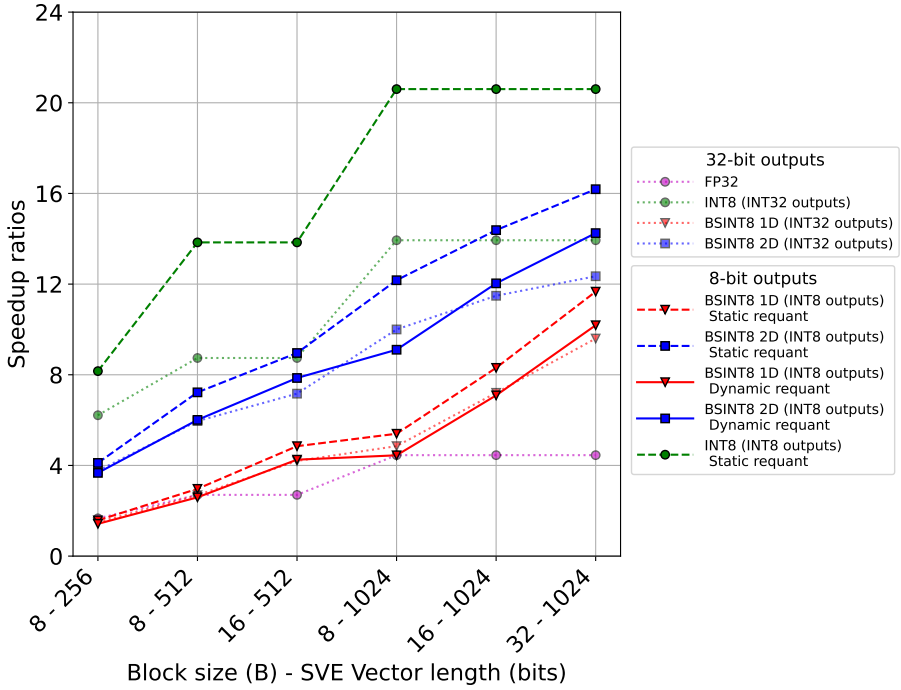


Figure 6.12: BS-INT8 requantization cycle-count speedup comparisons (All INT8 and BS-INT8 microkernels are *dp* type). Specific combinations of block size and vector lengths that enable fused operation are considered. Normalizations are with FP32 SVE 128 bits cycle counts.

memory transfers. Hence, we mainly look at kernels with block sizes feasible within the specific microkernel widths that enable fused requantization.

In Fig. 6.12, we evaluate speedups of several GeMM computations (with INT32 outputs) along with the inclusion of their requantization counterparts (INT8 outputs). It is very important to mention that requantizing the output not only helps in preparing the data for the next layers, but also reduces the data sizes that need to be sent to memory from registers which is another critical parameter for benchmarking speedups. This can be observed from the speedup curves for 2D BS-INT8 requantization kernels, which inevitably includes more computations than its non-blocked INT8 counterpart with INT32 outputs. However, at some points the speedup is found to be higher for dynamic requantization case which is predominantly due to the reduced output bitwidth benefits. To confirm this, if the INT8 computation output were truncated

to 8-bits before storing to memory, we can observe from the figure that the performance for INT8 becomes much higher. Hence, it is important to also consider the memory transfer footprint while exploring the GeMM kernel options. Furthermore, we can perform our computation without any FP SIMD/vector units which would additionally provide significant benefits in terms of HW area and power [7, 51].

6.5.6 Evaluation on CNN inference workloads

We explore the performance impacts of our BS-INT quantization on standard CNN benchmarks. Figure 6.13 showcases our CNN deployment studies, illustrating the cumulative instruction counts for running all GeMMs in the ResNet20-CIFAR10 model across different BS-INT8 and baseline datatypes. It also depicts the speedups obtained in each configuration normalized to our FP32 baseline. For static and dynamic requantizations, the 2D blocks can be up to ∼2x faster than 1D blocks, and have on average ∼3.8x and ∼2.9x higher speedups than FP32 models respectively for ResNet20 model on CIFAR10/100 datasets. We analyzed the test accuracies of pre-trained ResNet models on CIFAR10/100 datasets using post-training quantization methodology.

We firstly fold all batch norm layer parameters into the CNN weight parameters, and then replace each convolution with an unrolled GeMM operation. Then, we perform the simulated quantizations using custom libraries [247]. Fig. 6.14 shows the impacts of different BS-INT configurations. We firstly observe that the accuracy degradation is negligible for BS-INT8 models with 8-bit mantissas compared to FP32 models. While for BS-INT6 and BS-INT4 with 6 and 4 bit mantissas, the general trend as expected is decreasing accuracies with increasing block sizes. This further degrades for 2D activation blocks compared to 1D blocks indicating a potential trade-off opportunity, wherein low-precision MAC instructions can be optimally used with tunable block shape and size. Accuracies can potentially be further improved by adopting a quantization aware training methodology for the configurations that encountered severe degradation.

6.6 Conclusion

While the previous chapters focused on analyzing the sensitivity of diverse workloads to quantizations, specifically through BFP quantizations, there were no specific assumptions made on the target HW platforms, keeping the previous experiments open to different HW platforms with potentially different HW metrics such as throughputs and power consumption. However, in this chapter,

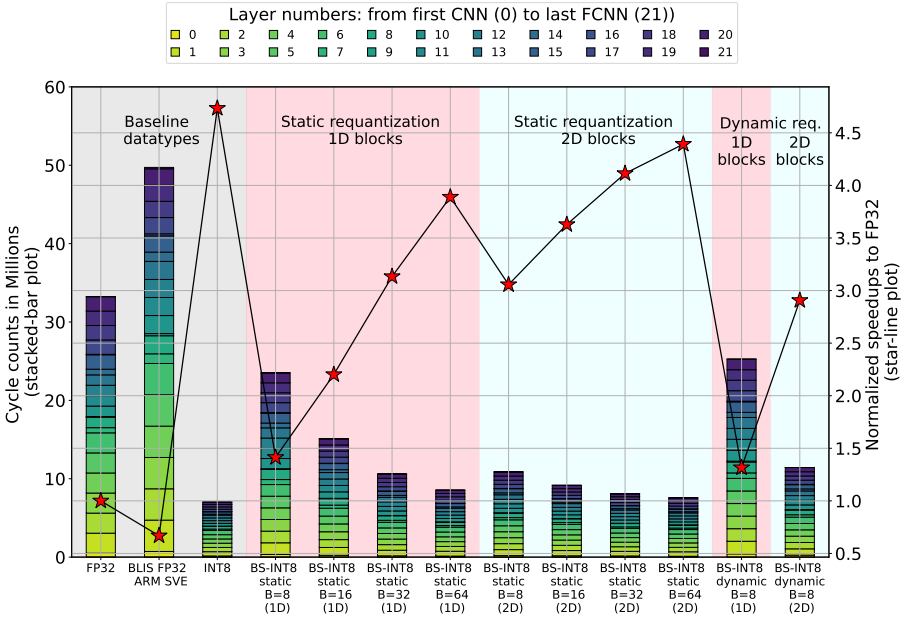


Figure 6.13: BS-INT8 GeMM kernel profiling for running ResNet20 CNN models and showing speedup comparison against other baseline datatypes (labels indicate layer number including downsampling layers). SVE configuration was set to 256 bits vector length. Note that for dynamic requantization, only scenarios with fused requantization are depicted. We can again observe that the speedups increase for configurations with larger blocks, 2D activation block shapes and static requantization. (Background colours representation: Gray colour – baselines; pink and cyan colour – Our BS-INT8 kernels with 1D and 2D blocks respectively).

our aim is to profile and benchmark block-scaled kernels on general-purpose vector instruction-set processors, so as to gain a more holistic view on the algorithmic and HW performances.

With this aim, we investigate the deployment opportunities for block-scaled datatypes on efficient, fully-programmable processors without any custom instructions. A design methodology is also presented to deploy optimized block-scaled integer GeMMs on scalable vector processors. We further optimize these kernels through design space explorations on multiple design parameters across processor vector lengths, block sizes, requantization techniques, and microkernel MAC instructions. We observe that longer vector lengths are generally beneficial for speedups, however can be redundant in some unrolling scenarios. Fused

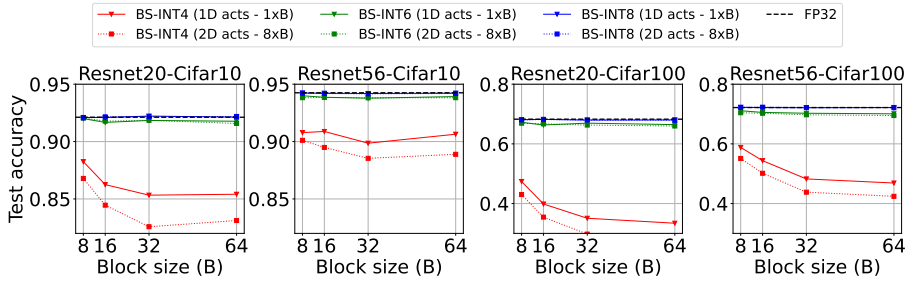


Figure 6.14: BS-INT quantization impacts on CNN test accuracies. Analyzed different quantization bit-widths and activation block shapes (1D and 2D).

dynamic requantization scenarios suffer some overhead, although they make the deployment easier, more robust and reduce the data transfer size from registers to the memory. Accuracy impacts of our tunable BS-INT kernels through flexible block size and shapes are also analyzed for CNN classification tasks. Utilizing 2D activation blocks instead of conventional 1D blocks, the static and dynamic BS-INT8 configurations yielded on average 3.8x and 2.9x GeMM computational speedups over FP32 models respectively, at no accuracy loss for CNN classification tasks on CIFAR10/100 datasets.

In the next chapter, we aim to move towards achieving more co-optimized solutions between algorithmic sensitivities and HW metrics, particularly for block-scaled DNN training process. Training workloads are especially interesting for block-scaled datatypes as they involve reusing the same matrices that were used in the forward pass, but in a transposed manner for the backward pass. We will additionally look at what algorithm-system co-optimizations are possible to eventually develop an efficient quantized training strategy with block-scaled kernels in terms of block shapes, considering the same target HW platform as in this chapter.

Chapter 7

Block-scaled GeMM with adaptive block shapes for DNN training at the edge

The previous chapter explained the different configurations of our block-scaled kernels and techniques for optimizing the kernels for meeting target requirements. In this chapter, we further explore the research question defined in section 1.2, *Q2. How do we efficiently deploy DNNs with block-scaled formats using general-purpose vector instructions,* specifically to perform more efficient DNN training?

The setup from previous chapter now enables us to compare the HW metrics for different block-scaled GeMM kernel deployments using general-purpose vector instructions of ARM SVE. We explore here the more challenging workload of performing DNN training at the edge. Block-scaled data formats have emerged as a popular HW-SW co-optimized choice for DNN training workloads on custom accelerators [47, 102, 118, 246] due to their increased dynamic range capabilities and efficient requantization at fine block-level granularity.

We perform further optimizations based on the fact that square blocks are transpose-compatible, i.e., the block compositions for a matrix are optimized for performing dot products in either the forward pass or in its transposed form in the backward pass. We also identify from the previous chapter's explorations that square, a special case of 2D blocks, has higher throughput than its counterpart 1D or vector block. However, the square blocks have larger equivalent block sizes than vector blocks, while ensuring that the number of alignments required for accumulations across different blocks in the GeMM

remain similar. Note that a comparison is possible where both configurations have equal block sizes, however this is not our objective as this complicates the kernel design and requires modifications to the programming model which is left for future work. Instead, here we are aware of the fact that square blocks would have higher quantization errors due to its coarse block configuration presenting a potential trade-off strategy. Consequently, we propose an adaptive training approach to move towards co-optimized solutions by initially performing HW-efficient coarse-commutative training with square blocks and switching to accuracy-driven fine adaptation with vector blocks in the final stages.

This chapter is based on the published conference paper: Nitish Satya Murthy, Nathan Laubeuf, Debjyoti Bhattacharjee, Francky Catthoor, and Marian Verhelst. *"Adaptive block-scaled GeMMs on vector processors for DNN training at the edge"*. In: 2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC) [140].

7.1 Introduction

Recent concerns regarding privacy, reliability, and latency have led to a growing interest in techniques to efficiently train DNNs at the edge [132]. This becomes even more essential for tasks that depend on real-time or online learning, for effective functionality. Continuous control tasks, as discussed previously in chapter 5, are such scenarios in which MBRL algorithms are used for desirable control demanding efficient DNN training through environmental interactions in the field to learn its dynamics model [38]. These tasks encounter additional challenges of resource constraints in scenarios where, for example, a robot is battery-powered and needs to operate in dynamic environments [148].

As described throughout the thesis, block-scaled datatypes are gathering increasing attention for efficient DNN implementations. They are especially attractive for training workloads since they offer increased dynamic ranges. Also, since offline calibration is not feasible for training when compared to inference, the dynamic requantization performed at runtime for block-scaled datatypes significantly reduces the deployment complexity. However, most reported implementations of block-scaled datatypes have so far required specific HW support [45, 118, 246].

Performing block-scaled quantization requires the selection of which and how many elements share a common exponent or scale. To maximize computational efficiency when using block-scaled DNNs, the block-quantization axis should align with the accumulation-axis in GeMM, as that enables efficient dot-product accumulations within each block. DNN training typically involves transpose and

quantization operators in the backward pass, in contrast to inferences which only require the forward pass. If these operators are non-commutative, block-scaled DNN training will result in extra block transformation overheads [147, 186], which we analyze later in section 7.3.1.

To achieve high accuracy within the latency and energy constraints of edge deployment, it is crucial to make the training process adaptable. Adaptive training for block-scaled datatypes has been considered in the past for dedicated ASIC accelerators, by means of varying bit-precision [246] and block sizes [118]. However, these methods cannot be adapted directly for deployment on general-purpose vector processors.

In this chapter, we introduce techniques to enable performing efficient DNN training using block-scaled datatypes on processors with only general-purpose vector instructions. Our contributions are listed as follows:

- We propose square block shapes as opposed to conventional vector shapes for block-scaled datatypes and implement GeMMs using them on ARM SVE-based processors.
- We develop an adaptive block-scaled DNN training approach, by initially performing "coarse-commutative" training with square blocks, followed by "fine-adaptation" using vector blocks, to subsequently allow accuracy-speed trade-off.
- Our proposed method obtained around 1.95× GeMM computational speed-up with up to 2× lower memory footprint for fully-connected (FC) NNs using adaptive training over baseline FP32 models.

The rest of the chapter is organized as follows: Section 7.2 discusses the background and related works. Section 7.3 explains our methods. Sections 7.4 and 7.5 describe our experimental setup and results respectively. Lastly, we present our conclusions in section 7.6.

7.2 Background and related work

7.2.1 DNN training at the edge

Enabling DNN training at the edge efficiently is currently a highly trending research topic, towards which multiple approaches have been explored [251]. [22] identifies activation memory as the main bottleneck during DNN training, and proposes to only update the biases of the network which do not require

intermediate activation storage. Additionally, to recover some accuracy degradation observed with only bias updates, they modify the network architecture by adding a trainable down-sampled bypass route which requires reduced activation memory storage for backpropagation updates. However, for more general optimizations, several works rely on optimizing the most computationally demanding part of training which are GeMMs through quantization by effective number representations. [125] focus on optimizing the training graph with INT8 operands by using a loss-scaling approach to reduce gradient quantization impacts and using sparse computations to only update the most important weight tensor sections. However, these approaches require HW-inefficient FP scale computations during the requantization process and require offline pre-calibration overheads. A more general approach was utilized by [142] to mainly optimize GeMM computations on RISC-V processors, while utilizing reduced-precision but still expensive FP16 formats. Block-scaled data formats are rising in popularity for performing efficient DNN training due to their increased dynamic range and efficient requantization with power-of-2 scales at fine block-level granularity [47, 102, 118, 246]. Most of the previous approaches involved custom HW implementations. Few works targeted programmable vector processors, but only evaluated tensor-level exponents [48] [20]. In contrast, we focus on extending the block-scaled GeMM kernels developed in the previous chapter, for DNN training through flexible block shapes and sizes utilizing only the available ARM-SVE instructions.

Quantized DNN training: Considering a DNN with layer L involving forward propagation of uncompressed activations $A^{[L]}$ and weights $W^{[L]}$, the following equations represent the computations involved during block-scaled training.

$$A^{[L+1]} = Q_{Vr}(A^{[L]}) \times Q_{Vc}(W^{[L]}) \tag{7.1}$$

$$G^{[L]} = Q_{Vr}(Tr(A^{[L]})) \times Q_{Vc}(E^{[L+1]}) \tag{7.2}$$

$$E^{[L]} = Q_{Vr}(E^{[L+1]}) \times Q_{Vc}(Tr(W^{[L]})) \tag{7.3}$$

Where, Q_{Vr} and Q_{Vc} are block-scaled quantizers like in Fig. 7.1 for vectors in the row and column direction respectively (quantization direction along GeMM accumulation axis), Tr is a transpose operator, L is the layer index, $\times$ denotes matrix-multiplication operator. Eq. (7.1) is the forward pass which is calculated end-to-end for all indices L, and all intermediate uncompressed $A^{[L]}$ are stored for later usage. Eq. (7.2) calculates the gradients $G^{[L]}$ for their respective weights $W^{[L]}$ using all stored $A^{[L]}$. Eq. (7.3) represents the error propagation proceeding in the opposite direction of Eq. (7.1). Note that the quantizer

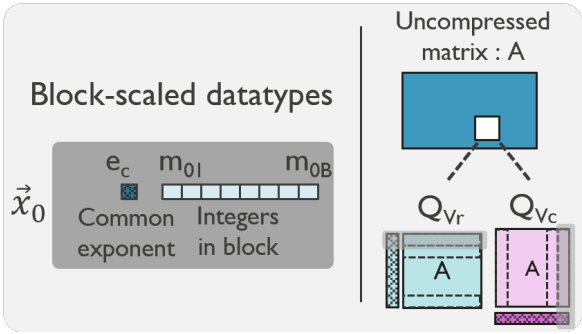


Figure 7.1: Block-scaled datatypes with 1D vector blocks (row/column) are demonstrated.

nodes enable the GeMM operations to be computed in efficient block-scaled computations.

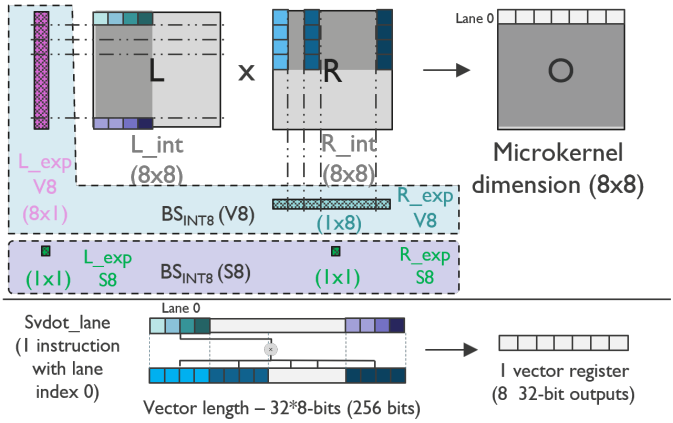
7.3 Block-scaled integer GeMMs for DNN training

7.3.1 Optimizing block shapes for DNN training

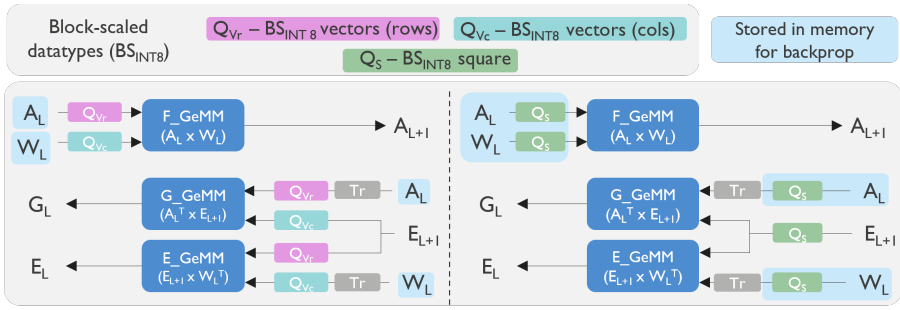
Fig. 7.2a illustrates an 8x8 microkernel calculation part of a bigger GeMM kernel. With BS-INT8 quantization, the matrix multiplication within a block involves integer dot-product operations on the mantissa's and a single shared exponent addition operation as shown previously in section 3.3, Eq. (3.9).

The integer matrix-multiplication part is mainly computed using lane-based dot product instructions that output 32-bit integers. With a vector block-scaling approach, to get optimized dot products in a GeMM requires the alignment of block-quantization and accumulation axes. That is, the left (right) matrix should involve row-wise (column-wise) blocks resulting in a column (row) exponent vector.

On the other hand, square blocks will have a single scalar exponent per block tile. Note that in the previous chapter 6, we mainly discussed 2D blocks, however here we present a special case of the 2D blocks which is a square. We propose utilizing square blocks instead of conventional vector blocks, due to their advantage of being commutative in quantization-transpose operations as illustrated in Fig. 7.3. It details how the operation orders of performing transpose and block-scaled quantization leads to different outputs for vector



(a)



(b)

Figure 7.2: (a) Illustration of matrix multiplication microkernel (8x8) with integer parts and exponent parts (V8 and S8 configurations). Note that each number and the common exponents are 8-bit integers. An example of lane-based dot product using ARM SVE instruction on a 256-bit vector processor is also illustrated. (b) The optimizations in DNN training computation graph after reversing the orders of quantization and transpose operators in backward pass for square shapes.

shapes (patch 1), while producing the same outputs for square shapes (patch 2). This allows us to reverse these operation orders for squares in backward passes, leading to optimization opportunities as shown in the training computation graphs in Fig. 7.2b.

Fig. 7.2b demonstrates that block-scaled activations ($Q_S(A^{[L]})$) and weights

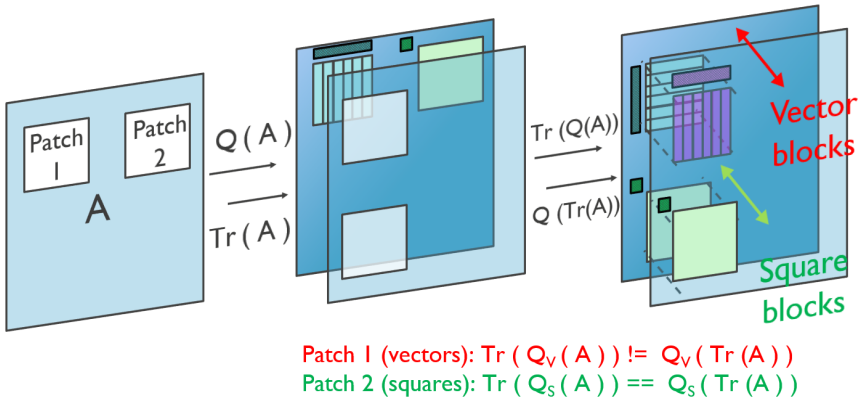


Figure 7.3: Investigating commutative property of transpose and quantization operators for different block-scaled quantizations.

$(Q_S(W^{[L]}))$ can be stored with their common exponent for later usage in the backward passes for square blocks. This leads to lower memory footprint and data transfer costs. Moreover, this circumvents having to perform extra quantizations before GeMMs by storing already block-quantized values. Vector blocks, on the other hand, would after transpose, lose their common exponent per element of the block, and hence cannot be stored in block-compressed formats requiring explicitly storing the exponent of each element. It is also to be noted that other shapes than square such as rectangles would also not have the same block elements with shared exponents in forward and backward passes. However, irrespective of the shapes, a similar concept that a block should be such that the same block elements are shared can be generalized to other computations such as convolutions and sparse matrix-multiplication computations.

Notations: In the rest of the chapter, we represent our block-scaled datatypes by BS-INT8 (V/S) for vector/square block shapes respectively. Block size is denoted by a number next to the shape. Ex.: BS-INT8 (V8) denotes vector blocks sharing an 8-bit exponent among 8 elements of INT8 numbers. BS-INT8 (S8) forms a square block of 8x8, sharing an exponent among 64 elements.

7.3.2 Vectorized BS-INT8 GeMM kernels for efficient computation on vector processors

While the previous subsection justified our square blocks and its impact on the DNN training complexity, there is additional impact on the GeMM deployments themselves. The mapping of the block-scaled types with vector instructions presents multiple challenges, since the instruction set extension does not natively support these formats [194]. Furthermore, auto-vectorization of the block-scaled computation kernels cannot be performed by an off-the-shelf compiler effectively demanding careful mapping for high utilization. The support for integer operands by the ARM-SVE extension can be effectively leveraged while computing in the block-scaled domain, as we demonstrated earlier in chapter 6, and again highlight in this section for GeMMs in a DNN training context. Figure 7.4 also shows the pseudocode of our BS-INT8 vectorized kernel implementations using ARM SVE vector instructions, along with the difference in instruction counts due to block shapes (parts highlighted in blue). Our block-scaled GeMMs involve an integer calculation part, and separate exponent handling computations for aligning exponents of different block partial accumulations.

The reduced GeMM instruction count for square versus vector shapes mainly arises from exponent handling overheads due to their different number of exponents for same matrix dimensions. The block size along accumulation axis directly relates to the alignment overhead needed for performing block-scaled GeMM kernels. It is important to note that the number of elements sharing a common exponent is not identical among the two shapes, rather the number of elements in the accumulation direction is kept similar instead. This leads to square shapes having higher compression ratios than vectors, leading to also lower instruction counts, however, at the cost of higher approximation errors as will be seen in section 7.5.1.

7.3.3 Adaptive DNN training through vectorized BS-INT8 GeMMs (vector and square blocks)

The previous subsections described the differences in HW metrics (instr. counts, memory footprint) due to different block-shaped kernels. However, these block shapes which have different compression ratios would also lead to different training convergences (SW metrics - loss/accuracy) due to different quantization errors. Hence, we propose an adaptive training approach here to obtain an optimized solution that properly trades off HW and SW metrics as illustrated

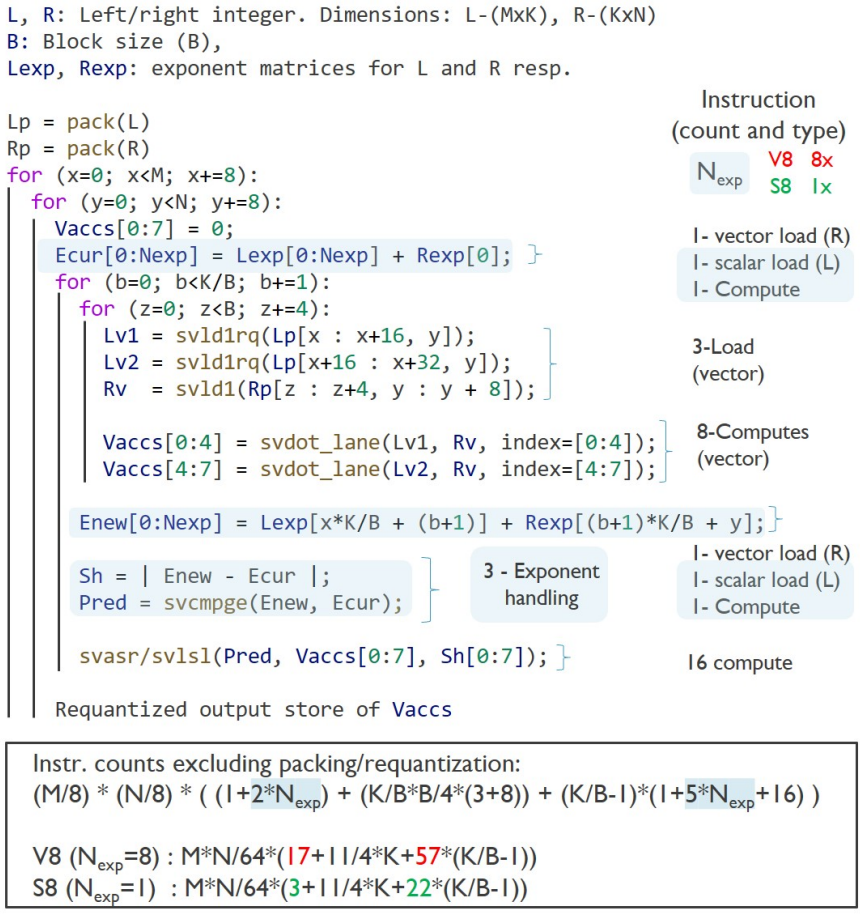


Figure 7.4: Pseudocode for our GeMM kernel using BS-INT8 formats with square and vector shapes, also highlighting the instruction counts for either scenario. Vector instruction details presented previously in chapter 6, Table 6.1.

in Fig. 7.5. As seen in the figure, the square blocks lead to faster convergence but low accuracies, while vector blocks exhibit the reverse trend.

Algorithm 5 details our adaptive approach to change block shape during the training process. A quantization configuration is set to first perform BS-INT8 (S8) training and then switch to BS-INT8 (V8). We define a hyperparameter α that is used to split the total training process across the 2 configurations. This follows the principle of performing a coarse training

Algorithm 5 Adaptive BS-INT8 DNN training

Conf_list: [BS-INT8 (S8), BS-INT8 (V8)], Max epochs: E_{max}, Epoch ratios with α: E_r=[α, 1-α], DNN model (weights W): M_W.
for id in [0, 1] **do**
  Model quantization set to Conf_list[id]
  if id==0 **then**
    M_W random weights initialization
  else
    $M_W[id] = M_W[id\text{-}1]$ (Initialize with prev weights)
  end if
  for t = 1 to $E_r[id]\cdot E_{max}$ **do**
    Train DNN on M_W with training data
    Monitor average loss, L.
  end for
end for

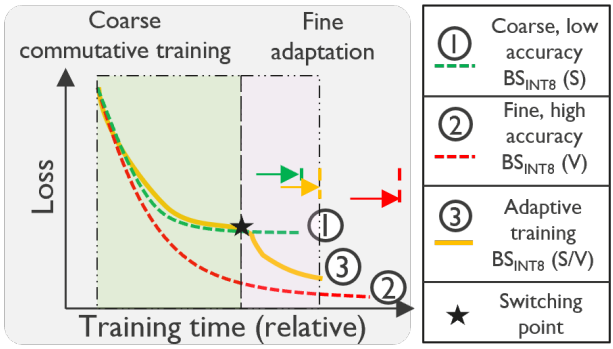


Figure 7.5: Adaptive training for accuracy-speed trade-off.

search that is more computationally-efficient (coarse-commutative) and later performing fine adaptation which is slower, but more accurate. As will be shown later in section 7.5.3, performing this multi-stage training helps in reaching optimized solutions based on accuracy-speed trade-offs.

7.4 Experimental setup

Workload: We analyze FC NN models which are trained to predict the system dynamics of continuous control tasks for benchmarking our approach. As

described in section 7.1, these tasks such as robotic control require real-time or online learning based on the agent interactions with the environment at the edge. Hence, efficient embedded training techniques would aid in optimizing task performance with limited resource constraints.

These robotics tasks involve FC NNs with state-action pair inputs – current robot state values and actions that control the robot. It outputs a probability distribution (mean and standard deviation) for the next state values capturing the uncertainty in its predictions, which is utilized for effectively controlling the robot based on user specifications [38]. While chapter 5 focused on optimizing the control performances, our aim in this chapter is instead on performing training efficiently. With this target, we analyze different training performances by simulating our BS-INT8 quantization configurations in PyTorch using quantization libraries [247]. The cartpole models use 3 layered FC NN with 500 hidden neurons (goal is to balance a pole in upright position). The pusher and reacher models use 3 layers with 200 neurons (goal is to push an object / reach a goal using a robotic arm). The halfcheetah model uses 4 layers and 200 neurons (goal is for the model to move/run fast). ReLU activations, an Adam optimizer with initial learning rate of 0.001 and a Gaussian loss function are used in all scenarios. This is mainly used as a demonstration task, while our technique is applicable to any generic DNN training task which involve GeMM operations. However, note that these models are relatively small when compared to large vision classification models or large language models, which might require additional optimizations on top of our block-scaled kernels, such as distributed computing on multi-cores, sparse updates with layer contribution analysis, etc.

Target HW details: We target general-purpose vector processors, in particular ARM v8.4a+ architectures with SVE extension because of its wide-spread usage in the edge and mobile community. Our experiments are run with the 256 bits SVE vector length. We benchmark our kernel runs through instruction/cycle counts on a gem5 [130] MinorCPU model with 64kB L1 cache and 1MB L2 cache memories.

Kernel code configurations: All matrices of GeMM operands are assumed to be stored in row-major formats. Furthermore, in BS-INT8 configurations, the exponent and integer matrices are stored separately. The code implementation is written in C using vector intrinsic functions from ARM C Language Extensions (ACLE) for high efficiency. ARM gcc cross-compiler 12.3 was used on x86-64 host platforms targeting aarch64 platforms.

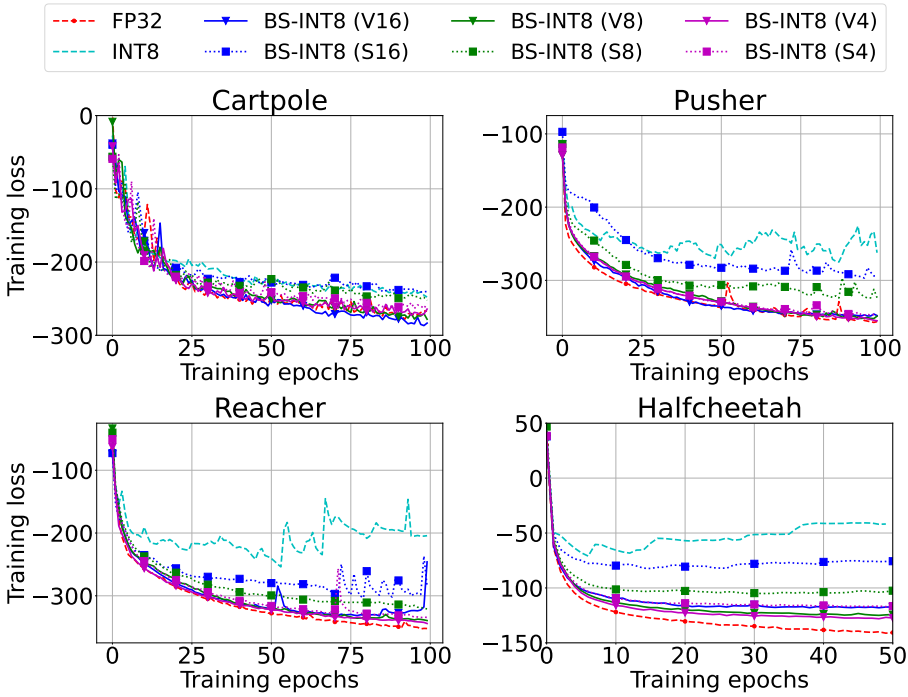
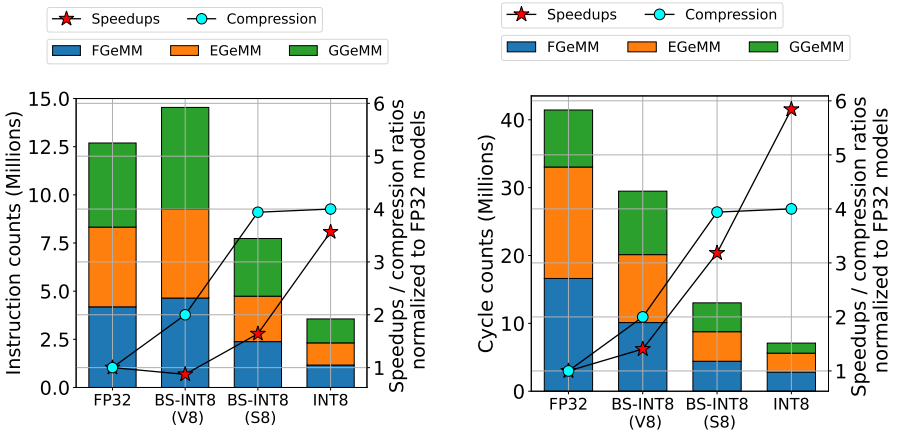


Figure 7.6: Impacts of block-scaled datatypes (shape / sizes) on DNN training convergence (Baseline models – FP32 and INT8 (tensor-level scale)).

7.5 Results and analysis

7.5.1 DNN training convergence due to block-scaled datatypes

We analyze here the impacts of block shapes (S/V) and sizes (4/8/16) on the training convergence quality for different datatypes. As observed from Fig. 7.6, the convergence is worse for square shapes due to more number of block elements that share a common exponent compared to vector blocks. The convergence quality also reduces for increasing block sizes (the largest block granularity is for INT8 with tensor-level scales), which is more prominent in the relatively more complex task - halfcheetah. Hence, although the square blocks are more suited for training computationally, it is still prone to larger quantization errors. This provides us with crucial design parameter choices that can be tuned based on the desired trade-off user requirements.



(a) Instruction counts.

(b) Cycle counts.

Figure 7.7: Vectorized GeMM kernel execution profiling for cartpole FCNN model with different datatypes depicting instruction/cycle counts, speed-up and memory compression ratios. Cycle counts are a better system-level metric and would hence be used hereafter for relative speed-up comparisons.

7.5.2 Benchmarking BS-INT8 vector and square GeMM kernels

While we have analyzed the training convergence impacts with BS-INT8 (S/V) formats, here we additionally examine the GeMM kernel deployment metrics. We explore the instruction and cycle count metrics for performing BS-INT8 GeMMs in different phases of backpropagation and also compare them with conventional datatypes (baseline - FP32 and INT8 formats). In Fig. 7.7, we can firstly observe the split-up of instruction and cycle counts for the cartpole model. Each operation phase (FGeMM for forward, GGeMM and EGeMM for backward gradient and error computations as in Eqs. (7.1), (7.2) and (7.3) respectively) require similar counts as they involve GeMMs of similar dimensions. Note that cycle counts are a more relevant system-level comparison metric as they include processor features such as pipelining, caching, etc., and would be used for further speed-up comparisons. We observe increasing speed-ups as we go from FP32, BS-INT8 (V8), BS-INT8 (S8) to INT8, while the quantization error metric for these datatype also increases accordingly. This speed-up increase from V8 to S8 is mainly due to the reduced overheads of exponent handling as seen earlier in the pseudocode in Fig. 7.4. The figure also shows the compression ratios leading to reduced memory footprints and data transfer costs, while handling intermediate activations during backpropagation. Note that BS-INT8

(V8) has only 2× compression, as we need to store exponents for each element as explained in earlier sections. Lastly, these flexible deployment choices arise from vector processor programmability while not needing any extra custom instructions, and yet providing different options for optimized HW-SW trade-off.

7.5.3 Adaptive DNN training through vectorized BS-INT8 GeMMs (vector and square)

We observed in the previous sections the trade-off between BS-INT8 (V8) and BS-INT8 (S8) in terms of HW instruction/cycle count metrics and SW training loss metrics. In this section, we showcase how we exploit this trade-off with an adaptive approach. Compared to baseline FP32 models, our non-adaptive vector block models achieve 0.9 – 1.4× faster GeMM computations, while the square block models are 2.1 – 3.2× faster across different tasks. Baseline INT8 models with tensor-level exponents are efficient but suffer significantly in convergence as observed previously in the loss curves in Fig. 7.6. Figure 7.8 also shows the results of our adaptive approach that follows Algorithm 5 illustrated earlier in Fig. 7.5 through a tunable parameter α to decide the ratio of training epochs performed in BS-INT8 (S8) vs BS-INT8 (V8). This enables us to reach more co-optimized solutions than non-adaptive approaches, highlighted with blue box. For $\alpha = 0.95$, we observe very close speed-ups to BS-INT8 (S8), while ensuring no significant accuracy loss for most tasks. The trade-off is almost observed to be mostly flat with a sudden change for square configurations, indicating that very few adaptations with vector blocks might be sufficient to retain algorithmic performance while the rest can be performed more HW-efficiently with square blocks.

7.6 Conclusion

DNN training at the edge induces many challenges due to resource constraints. These are addressed by efficient number representations such as block-scaled datatypes. With this target, in this chapter, we propose the use of square block shapes to exploit their commutative property of quantization and transpose operators, which allows more optimized DNN training. We implement efficient square and vector shape block-scaled (BS-INT8 (S/V)) GeMM kernels on vector processors by only using available vector instruction sets (ARM SVE). Furthermore, we propose the adaptive combination of different shapes during training process to effectively trade-off HW metrics such as cycle counts and memory footprint, with SW training loss metrics. Benchmarking on FC NNs, our

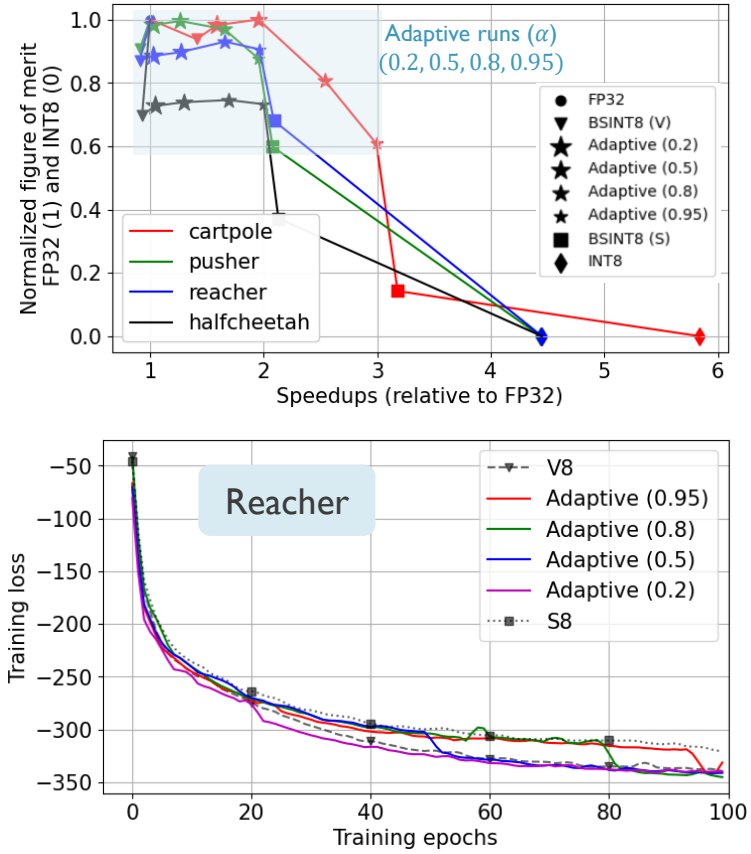


Figure 7.8: HW-SW trade-offs through our various BS-INT8 training configurations (in comparison to baseline FP32 and INT8 models) (Top figure). Training curves for different adaptive training ratios is also showed for the reacher task here (Bottom figure).

training approach can yield around 1.95× faster training GeMM computations with 2× lower memory footprint, compared to conventional FP32 models while reaching similar training performances.

Previously in chapter 6, we explored the design and optimization of block-scaled GeMM kernels using vector instructions in ARM SVE ISA. We further analyzed in this chapter, how these kernels could be used and optimized through different block shapes to enable efficient DNN training highlighting different trade-off

potentials. However, additional optimizations are achievable by moving towards domain-specific instructions, developed for block-scaled DNN deployments. Consequently, our objective in the next chapter is to focus on effective HW-SW DNN co-design on specialized SoftSIMD processors [242] through block-scaled datatypes, highlighting additional deployment strategies such as sub-byte instructions which could not be exploited earlier using ARM-SVE ISA.

Chapter 8

Domain-specific instruction-set processors towards more efficient block-scaled GeMM computation

The previous chapters 6 and 7 explored the block-scaled kernel design and optimization using general-purpose vector instruction-sets. In this chapter, we investigate the research question defined in section 1.2, *Q3. How do we increase efficiency for block-scaled DNN deployments by moving towards domain-specific instruction set processors?*

While previously, we showcased different co-optimization strategies enabling flexible deployment strategies with various block shapes and sizes, the precisions were limited to INT8 due to the constraint of available instructions. On the other hand, optimized DNN deployments can also exploit sub-byte quantization strategies effectively. There are few works that also investigate using general-purpose INT8 instructions to implement sub-byte instructions via efficient packing instructions to emulate the same MAC behaviour [100, 230]. While they do exploit increased subword parallelism due to reduced bit-widths, there exist overheads due to the emulation behaviour limiting overall performance improvement. In this chapter, we utilize a software-defined single instruction

multiple data (SoftSIMD) processor that enables flexible and energy-efficient mixed-precision computations with high vector utilization [242]. We explore how block-scaled integer DNNs can be efficiently deployed on such domain-specific instruction set processors. We also avoid expensive FP32 accumulators and instead use gradual precision increase as the accumulation depth increases to avoid overflows. While BFP and BS-INT have the same data representations, we refer to our work as BS-INT throughout the thesis to distinguish the scenario when we consider inter-block accumulations without conversion to FP32, which is common in BFP computations. Diverse DNN inference deployments with BS-INT quantizations are analyzed with different blocksizes, varied weight and activation precisions. We demonstrate clear trade-off opportunities with HW metrics such as energy and area efficiency while not degrading SW accuracy metrics too much compared to baseline datatypes.

This work has been done in collaboration with Pengbo Yu[1] who developed the SoftSIMD processor [242], but here we specifically target block-scaled integer (BS-INT) deployments. The details of HW design, HW performance and mapping method are from Pengbo Yu, and will be published in details as part of his PhD dissertation in 2025. All the other parts belong to a collaborative paper of Pengbo and the author, and will be submitted in 2025. All use of his work here has been acknowledged and permitted. The author's contributions include the quantization methodology, block configurations and algorithmic evaluation of block-scaled DNNs that enable efficient mapping on SoftSIMD architectures.

8.1 Introduction

Block-scaled integers (BS-INT), as we detailed in previous chapters, combine the wide dynamic range of FP representation with the low HW cost of FXP arithmetic. Recent work, such as MSFP [45], VS-Quant [43], FAST [246] and DBPS [118], have demonstrated the benefits of block-scaled data types for DNN inference and training. Such shared-exponent number representations are especially useful for maintaining good algorithmic performance even with low-precision quantization (<8 bits), without the need of any extensive and complex algorithmic techniques.

The key observation driving this chapter's contribution is that:

- Although block-scaled quantization at different precision levels has been explored, the precision supported by the HW is often determined at design

[1] PhD student at EPFL under the supervision of Prof. David Atienza

time and confined to specific values like 4-bit, 8-bit, and 16-bit. As a result, only specific mantissa precisions can be used at runtime, restricting its full potential. Additionally, the flexibility of mixed-precision configurations of different DNN operands is limited.

- Accumulation operation still requires the conversion of BS-INT values to FP and the use of the resource- and energy-intensive FP32 adder, as utilized by MSFP [45], FAST [246] and DBPS [118].

To tackle these challenges and opportunities, this chapter presents a novel microarchitecture for BS-INT computation that efficiently supports 8-bit exponent and variable mantissa precision without the need for FP accumulators. It employs SoftSIMD, which uses guardbits to partition subwords in the datapath, enabling flexible precision definition with high data-level parallelism at runtime, such as 3, 4, 6, 8, 12, 16, 24 bits mantissas. Moreover, a lightweight data pack unit is introduced to seamlessly bridge operands across different precision levels, allowing the mantissa to be repacked to a larger size during accumulation to avoid overflow and ensure accuracy. Thus, it can perform vector addition/subtraction/shift, vector-by-scalar multiplication, and in turn multiply-accumulate (MAC) operation. These instruction supports help in accelerating workloads such as DL algorithms and any general data-parallel workload. This design supports operations from low-to-high precision quantization. Compared to these formats, our proposed design achieves significant area reductions per MAC and improves energy-efficiency (TOPS/W) in CNN inference while maintaining similar accuracy.

The major contributions of this work are as follows:

- We introduce a novel microarchitecture that efficiently handles variable-precision BS-INT arithmetic, enabled by flexible subword partitioning through SoftSIMD.
- We present a strategy to progressively increase mantissa precision during BS-INT accumulation operations by using a data packing unit. This method prevents potential overflow, ensures accuracy, and eliminates the need for resource-intensive FP32 accumulators.
- Our design reduces per MAC area across various precision levels comparatively by up to a maximum of 92% over FP32, 86% over bfloat16, and 53% over INT8 baseline implementations respectively.
- CNN inference experiments reveal that our design delivers up to 7.7x and 5.5x improvements in TOPS/W over FP32 and bfloat16, respectively, and even a 1.5x improvement over INT8, while reaching a similar baseline accuracy.

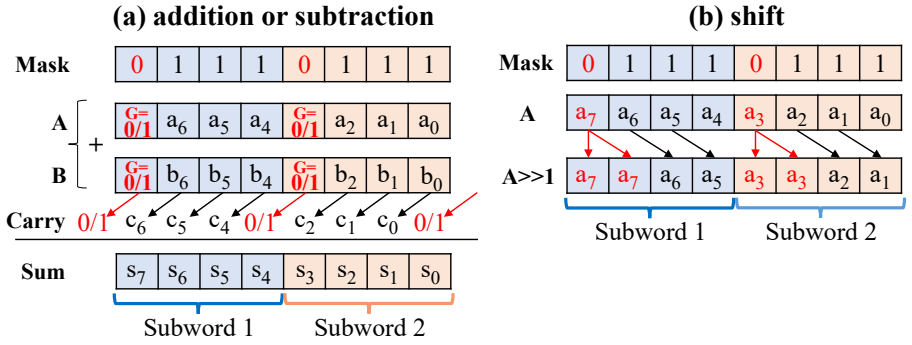


Figure 8.1: SoftSIMD (a) addition or subtraction, (b) shift.

8.2 Background

8.2.1 Software-defined SIMD

SIMD is a widely used technique for exploiting data-level parallelism to enhance efficiency, but most HW supports only a limited set of modes, such as 4-bit, 8-bit, and 16-bit. In contrast, SoftSIMD can define arbitrary subword sizes at runtime using the mask vector and perform operations including vector addition/subtraction/shift and vector by scalar multiplication [24, 242].

In fixed-point addition and subtraction, N-bit operations can produce a (N+1)-bit result. Therefore, only N-1 bits can be used to avoid overflow, allowing the N_{th} bit to act as a temporary guardbit during computation. As shown in Fig. 8.1(a), the MSB bits of the two operands are set to '0' during addition to avoid overflow into the next subword and set to '1' during subtraction to ensure a carry of +1 into the next subword since A-B = A + $\bar{B}$ + 1. An XOR operation is later used to check and correct the MSB output. For the shift operation presented in Fig. 8.1 (b), the MSB bits perform a sign extension while the remaining bits do regular shifts. Similarly, vector by scalar multiplication can be decomposed and executed through iterative shift-add operations.

In this way, SoftSIMD allows dynamic configuration of the datapath across different precision levels while consistently maintaining high data-level parallelism. For example, a 48-bit datapath can be configured as 16×3-bit, 12×4-bit, etc.

Consequently, the work we present offers flexible, run-time-adjustable mantissa precision while ensuring accumulation can be done without conversion to FP, maintaining accuracy, and preventing overflow. Although, in our ARM-SVE

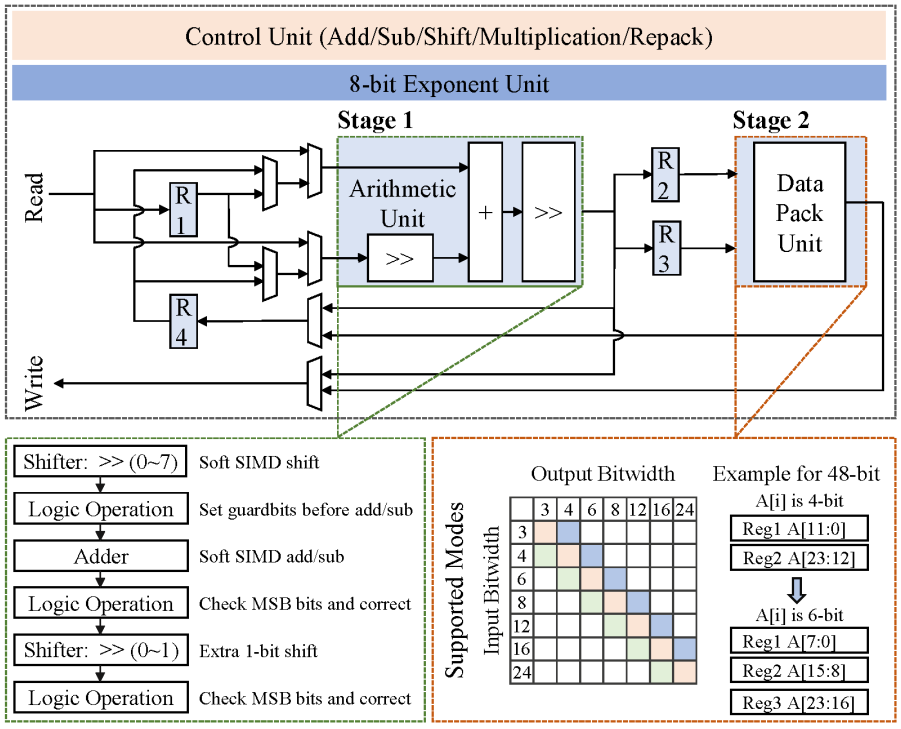


Figure 8.2: SoftSIMD microarchitecture overview.

implementation defined previously in chapter 6, we also consider accumulations without FP32 conversion, the deployment lacks flexibility and is restricted to a minimum operand precision of INT8.

8.3 BS-INT computing microarchitecture

8.3.1 Microarchitecture overview

The microarchitecture depicted in Fig. 8.2 extends the SoftSIMD approach [242] to support BS-INT arithmetic. Each datapath unit includes a dedicated 8-bit adder for exponent summation and a shift decoder for aligning BS-INT accumulations. The BS-INT mantissa format utilizes two's complement rather than sign-magnitude representation, ensuring better consistency in addition and subtraction operations while reducing HW resource usage. It consists of

two pipeline stages: the first-stage arithmetic unit and the second-stage data pack unit. The R1 to R4 registers are used for temporary storage and data reuse during computation.

The first stage includes a logarithmic shifter, an adder, a 1-bit shifter and peripheral logic gates, and it can support vector addition, subtraction, shift, as well as vector by scalar multiplication. The logarithmic shifter supports up to 7 right shifts per cycle, which is sufficient for the multiplication and accumulation of BS-INT discussed in Section 8.5.1. The extra 1-bit shifter is used to make sure that the MSBs are correct during shift-add iterations.

The second stage converts operands via repacking to a larger, equal, or smaller precision. It can be clock-gated when not in use and activated only as needed. In this work, twelve repacking modes are supported from 3 to 24 bits, as shown in Fig. 8.2. During BS-INT accumulation, it gradually increases the mantissa size, ensuring that there is no overflow and maintaining accumulation accuracy.

The datapath width is 48 bits or integer multiples of it, ensuring full utilization of the word space for the given precision levels in Fig. 8.2. The control path generates signals for addition, subtraction, shift, and data pack. For multiplication, it converts the multiplier into shift-add sequences and sums the exponents of the multiplicand and multiplier. For accumulation, it adjusts mantissa size through repacking signals and manages exponent alignment with shift control.

8.3.2 BS-INT arithmetic operation

The flexibility of SoftSIMD enables this microarchitecture to inherently support variable-precision fixed-point operations, including addition, subtraction, shift, and multiplication. For BS-INT multiplication, the multiplicand operand is a vector word containing multiple subword values within a block that all share the same exponent, and the multiplier is a scalar operand from another block. For models like CNNs, the multiplicand typically corresponds to the larger-size activation values from the input feature maps, while the multiplier represents the smaller-size weight values from the filters. The multiplication result is obtained by multiplying their mantissa and adding their exponents. Since their mantissa are in two's complement and restricted to the range (-1,1), the result remains within the original range, hence eliminating the need for exponent adjustment and mantissa right shifts. Therefore, it is fully compatible with the native multiplication support of SoftSIMD in fixed-point. Additionally, by using shift-add multiplication, operands with different bitwidths for the multiplier and multiplicand are allowed. A smaller multiplicand bitwidth results in higher data-level parallelism, while a smaller multiplier bitwidth reduces the shift-add

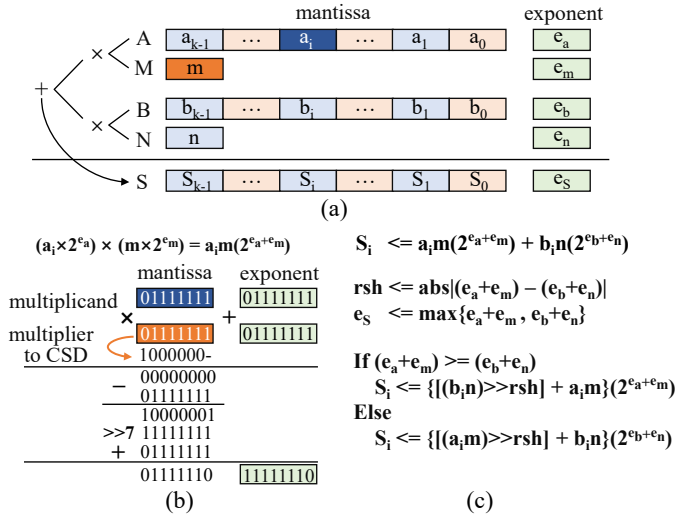


Figure 8.3: SoftSIMD arithmetic with BS-INT (a) sum of two multiplications, (b) shift-add-based multiplication, (c) addition between two BS-INT operands.

cycles, thus lowering multiplication overhead. In DNNs, this allows for the optimization of both activations and weights.

For BS-INT addition/subtraction between two vector operands within the same block, exponent alignment is not required. Similarly to fixed-point SoftSIMD, the mantissa of two operands is extended to a larger size via the data pack unit to ensure precision and prevent overflow, while the result's exponent remains unchanged. Thus, addition within the same block degrades to a fixed-point operation. For operands from different blocks, such as partial block accumulation results of a MAC operation, our design supports up to a 7-bit right shift and addition in a single cycle. If the exponent difference is less than or equal to 7, the addition can be completed in one cycle. Otherwise, the mantissa of the smaller operand is right-shifted and added with zeros until aligned, after which the addition is performed. However, this case is very rare in practice, as discussed in Section 8.5.1. Also, note that a large exponent difference between two operands in lower-precision cases often allows one operand to be approximated as zero.

Figure 8.3 illustrates how this microarchitecture performs a BS-INT MAC operation. In this case, operands A and B are vectors, where multiple mantissa in subwords share a common exponent, while M and N are scalars. For the multiplication between A and M, the exponents e_a and e_m are added, while the mantissa of M is converted into a sequence that controls the iterative shift

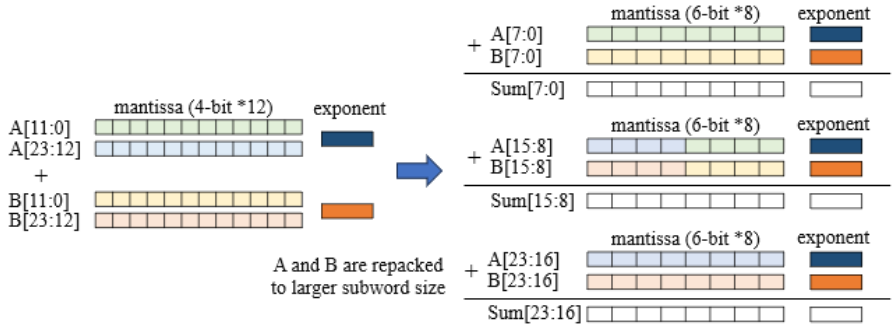


Figure 8.4: Example of accumulation precision increase during addition with a 48-bit datapath. Output sum is stored in 3 registers, and one of the operands is shifted before addition based on the difference between the exponent values.

and addition of A. Specifically, M is encoded using Canonical Signed Digit (CSD) [162], simplifying consecutive ones into a more efficient representation using '1', '0', and '-'(-1). For example, "0111"(1+2+4) becomes "100-"(8-1). This reduces the average shift-add cycles for an X-bit multiplier from X to X/3. Additionally, the shift-add operations start from the LSB and move toward the MSB of the multiplier, using the right shift to maintain the same precision as the multiplicand. The multiplication between B and N employs the same approach. The accumulation of the two multiplication results is performed similarly to the previously explained BS-INT addition. Note that it is necessary to increase the mantissa precision during accumulation to prevent overflow. An example of this is shown in Fig. 8.4, where the input operands are repacked from 4-bit to 6-bit during accumulation with the final output sum being stored in 3 registers.

8.4 Algorithms quantization and mapping

8.4.1 BS-INT quantization for inference

While we have flexible HW, it needs to be explored how we can utilize this effectively to reach co-optimized performances. Two key parameters are required to define the BS-INT configuration: 1) The block of values that share the exponent (block granularity). 2) The precision of the mantissa to be used within each block. The exponent is fixed to 8 bits to ensure sufficient dynamic range and maintain direct compatibility with FP32 values.

For our BS-INT quantization flow, we firstly obtain pre-trained FP32 model weights. We fold all batchnorm layers into corresponding previous layers [145]. We then apply post-training BS-INT quantization to determine sensitivities to different configurations. Lastly, for each BS-INT configuration, we perform minimal fine-tuning (1-10 epochs) in a quantization-aware manner [45].

The following configurations are explored for analyzing the inference performances of our experiments:

- Mantissa bits: 3 / 4 / 6 / 8 (weight / activations).
- Block granularity for activations (fine to coarse): Multiple rows (1/2/4) along channel-depth, per-channel.
- Block granularity for weights (fine to coarse): Multiple channels (1/2/4) along channel-depth. While multiple rows were also possible for the weights, the block size would become too small. Hence, we increase it to channel granularity.

Illustration of these granularities and their mappings are discussed in the next section 8.4.2.

8.4.2 Mapping to BS-INT

Since the proposed design allows vector addition and vector by scalar multiplication, we need to map the algorithms into the vector by scalar mode.

Figure 8.5(a) illustrates an example of per-channel BS-INT granularity in a convolutional layer, where each activation input channel and weight channel is treated as a block. The values in each activation block are unrolled into a $W_O \times H_O$ row vector with multiple subwords by im2col operation [28], forming a total $(W_O \times H_O) \times (k^2)$ row array. Similarly, each weight block is unrolled into a $(k^2) \times (1)$ column vector. Note that here we use vector by scalar mapping instead of vector by vector after im2col to be compatible with the proposed HW. The multiplication is performed between each row vector $(W_O \times H_O)$ and its corresponding scalar (1) in the column vector. The multiplication results within each block are accumulated by repacking and summing each row vector, producing a partial accumulation result. Then, partial results from different blocks are aligned by the right shift to match the exponents, then repacked as needed and summed to produce the final accumulation result. The mapping of fully-connected layers is also performed in a similar approach.

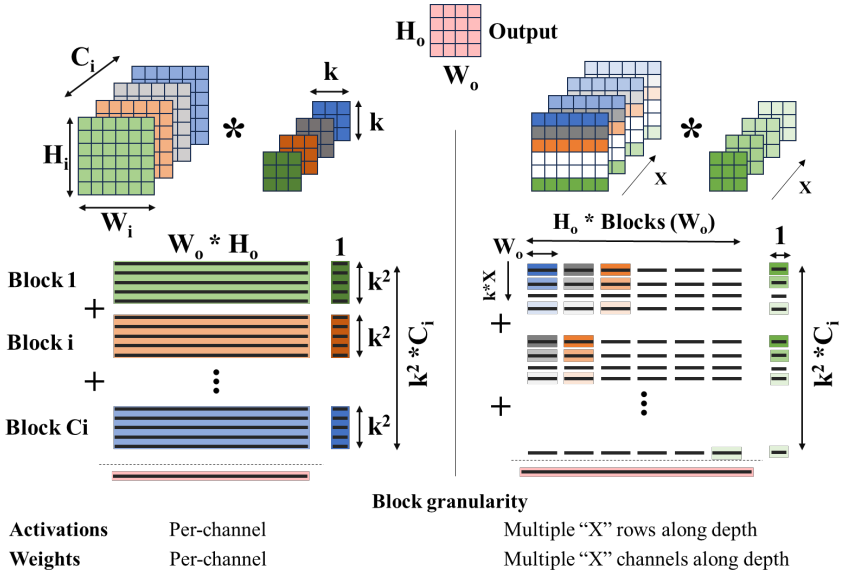


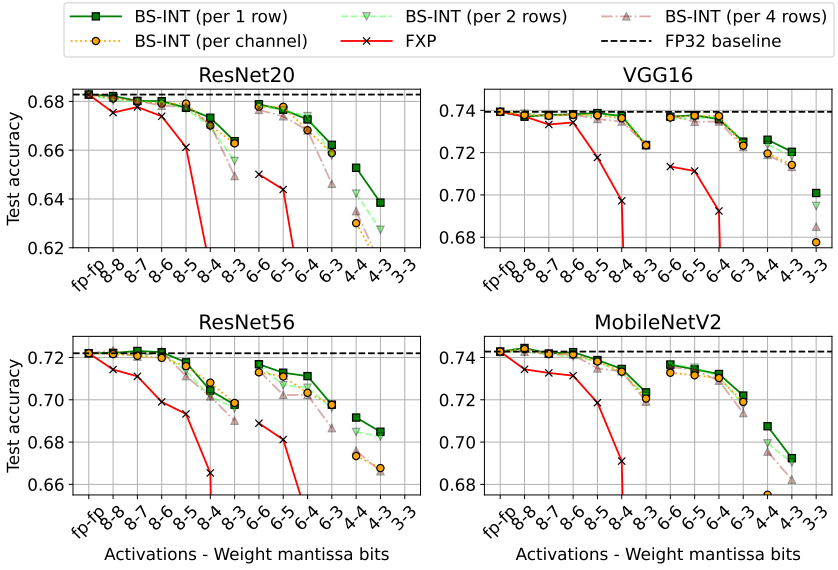
Figure 8.5: Map BS-INT with SoftSIMD with different block granularities: (a) per-channel blocks and (b) multiple rows/channels.

Figure 8.5(b) illustrates an alternative block granularity where one or multiple row vectors (X) along the depth are considered as a block, providing finer granularity. In this way, the mapping strategy is the same. The only difference is that each row vector is reduced from $W_O \times H_O$ to $W_O \times 1$, and the block size of the unrolled activations are decreased from $(W_O \times H_O) \times (k^2)$ to $(W_O \times 1) \times (kX)$, that X is the number of row vectors.

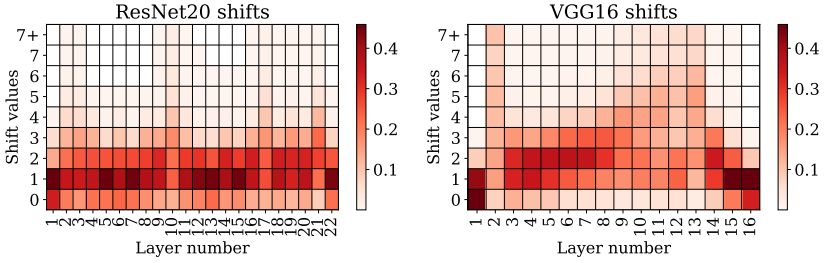
8.5 Experimental setup

8.5.1 Selected benchmarks and accuracy evaluations

We evaluate the quantized BS-INT performance on a standard image classification task – CIFAR100 dataset [108], and 4 popular CNN models – ResNet20 [82], ResNet56 [82], VGG16 [199], and MobileNetV2 [192]. As a baseline fixed-point comparison, we consider quantized models with per-tensor quantizations and power-of-2 scaling factor for effective HW computations [145]. Our flexible block quantizations are simulated using cuda libraries [247] in the PyTorch framework. Figure 8.6a depicts the impacts on test accuracies



(a)



(b)

Figure 8.6: (a) Test accuracies for mixed-precision quantized models with respective baseline FP32 accuracies (dotted lines), along with the impacts of different block granularities for activations. (b) The distribution density of accumulation shifts for ResNet20 and VGG16 models (6-bit activation and 4-bit weights with per-channel configuration).

for several mixed-precision BS-INT explorations. In all our experiments, the first and last layers always use higher precision (8-bits). To exploit the high flexibility of HW, we explore different mantissa bits for weights and activations. The figure also portrays the impact of block granularities (mentioned previously

in Section 8.4.1) in addition to different precisions. In general, block size and accuracy have inverse impacts, with, for example, BS-INT (per 1 row) having higher accuracy than BS-INT (per 4 rows) and BS-INT (per channel).

It can be observed from Fig. 8.6a that even for BS-INT configurations of 6-bit activations and 4-bit weights, we reach similar accuracies as baseline FP32 models ($\leq$1% accuracy drop). In particular, for this configuration, the distribution of shifts required during accumulations for ResNet20 and VGG16 models are illustrated in Fig. 8.6b. A significant amount of shifts is found to be lower than 7, which can be performed in single cycle, indicating low overheads for exponent handling.

8.5.2 Hardware implementation

We validated the proposed design using a 28nm process library with a standard voltage of 0.9V and a design frequency of 600MHz. The data path unit used here is 48 bits, supporting an 8-bit exponent and mantissa precisions of 3, 4, 6, 8, 12, 16, and 24 bits, with seamless conversion between adjacent mantissa precision. The design was synthesized using Synopsys tools and a power model was established through simulation of the synthesized netlist to evaluate energy consumption.

8.5.3 Baseline setting

To evaluate the proposed design, we designed and synthesized the alternatives FP32, bfloat16, and INT8 under the same conditions, which are widely used computation formats. FP32 implementation supports 32-bit floating-point operations and is compatible with all lower-precision mantissa. Bfloat16 implementation supports bfloat16 multiplication and is compatible with mantissa precision below 8 bits, while its accumulation needs FP32 to avoid overflow and precision loss. The INT8 implementation features an 8-bit multiplier and a 32-bit adder for accumulation, supporting specialized 8-bit fixed-point operations. Notably, this INT8 baseline avoids using an INT32 multiplier for scaling, instead employing optimized power-of-2 shifts for HW-efficient scaling.

In our design, one HW unit can not only support multiplication but also accumulation while the baselines cannot. For a fair comparison, our design also uses the same topology as the baselines, that is composed of one multiplication unit and one accumulation unit.

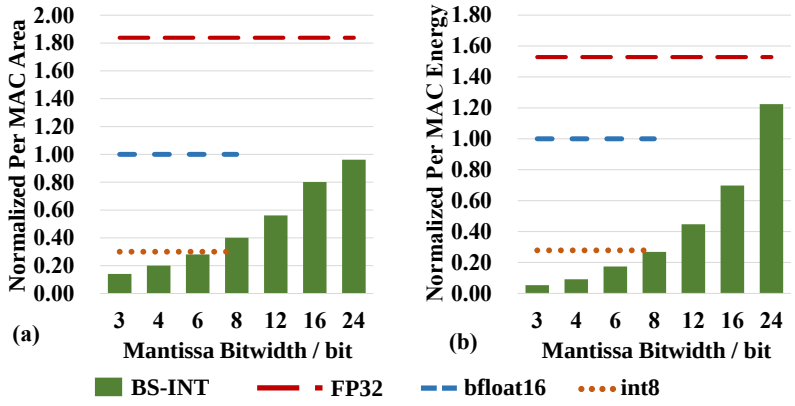


Figure 8.7: Normalized per MAC (a) area and (b) energy.

8.6 Results

8.6.1 Area performance

Figure 8.7(a) presents the normalized per MAC area of our proposed BS-INT design across different mantissa precision levels, compared to FP32, bfloat16, and INT8 alternatives. As FP32 consumes significantly more HW resources, bfloat16 is used as the normalization baseline, providing a clearer comparison.

For our BS-INT design, the n-bit MAC operation means multiplication in n-bit precision and accumulation in the closest supported higher precision, such as multiplication in 6-bit and accumulation in 8-bit. For baselines, the MAC precision is fixed according to their specific HW design.

Note that, as mantissa precision decreases, the benefit of our BS-INT design in area efficiency increases due to its flexible subword configuration/high parallelism and seamless switching between multiplication and accumulation operations. Specifically, when the operand mantissa precision varies from 8 to 3 bits, our BS-INT design achieves a reduction of 78%–92% per MAC area over FP32 and 60%–86% over bfloat16, respectively. As for the INT8 format, our design can reduce per MAC area by 53%, 33% and 7% for 3-bit, 4-bit and 6-bit respectively, while there is a slight increase for the 8-bit case, since our design needs data pack unit (2 per MAC unit) and local registers (8 per MAC unit) to provide more flexibility as well as support exponents.

When the mantissa precision exceeds 8 bits, we obtain higher precision than even bfloat16 and INT8 datatypes. However, our design can still reduce per MAC area by 48%–69% compared to FP32.

8.6.2 Energy performance

Similarly, Fig. 8.7(b) shows the energy per MAC operation normalized by bfloat16.

When mantissa precision decreases from 8 bits to 3 bits, our BS-INT design reduces per MAC energy by 82%–97% compared to FP32 and 73%–95% over bfloat16. Against the INT8 implementation, it reduces per MAC energy by 37%–81% from 6-bit to 3-bit precision, while with 3% at 8-bit. In this case, both the multiplicand and multiplier are 8-bit. Our design supports flexible multiplier size, where a smaller multiplier bitwidth reduces shift-add cycles, leading to lower energy consumption. For instance, for the 8-bit $\times$ 4-bit MAC case, our design achieves a 34% energy reduction per MAC.

Bfloat16 and INT8 formats can not support operand precisions beyond 8 bits, while our BS-INT design can still reduce 20% to 71% per MAC energy.

8.6.3 Inference performance

Finally, we map the CNN benchmarks to the proposed BS-INT design and three baselines, evaluating their energy consumption of all fully convolutional and connected layers.

As shown in Fig. 8.8(a), we present the normalized TOPS/W and inference accuracy for our design with per-channel and 1-/2-/4-row vector BS-INT schemes under 6-bit activations and 4-bit weights, alongside these of the baselines. Leveraging BS-INT's fine granularity of blocks, the accuracy loss of the per-row vector scheme is less than 1% relative to FP32 and bfloat16 and comparable to INT8. As block size increases from 1-, 2-, and 4-row vectors to per-channel, accuracy drops slightly while energy-efficiency improves, driven by enhanced vector unit utilization.On average, our design exceeds a 7.0x increase in TOPS/W compared to FP32, a 4.7x improvement compared to bfloat16, and a 1.3x gain in energy efficiency over INT8 baselines (per-row case).

Also, in Fig. 8.8(b) we present the detailed TOPS/W and accuracy versus mixed precision for VGG16 with the per-channel BS-INT scheme. As activations and/or weight precisions reduce, TOPS/W improves due to higher data-level parallelism and larger vector unit utilization, while accuracy drops slightly. In this example, with 6-bit activations and 4-bit weights, accuracy decreases by only 0.2%, while achieving TOPS/W improvements of 7.7x over FP32, 5.5x over bfloat16, and 1.5x over INT8 baselines.

Our proposed BS-INT design allows for flexible adjustment of mantissa precision

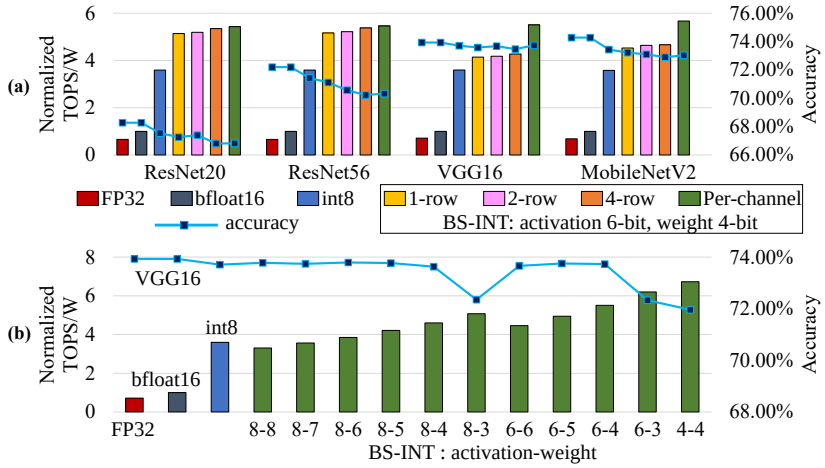


Figure 8.8: Normalized TOPS/W and accuracy for (a) different BS-INT schemes with fixed activation-weight precision of 6-4 bits and (b) different mixed-precision configurations with per-channel block granularity.

for activations and weights without compromising accuracy. It demonstrates clear advantages over FP32 and bfloat16 formats, and achieves comparable or better performance than INT8.

8.7 Conclusion

In this chapter, we shifted focus from general-purpose vector instructions to more domain-specific architectures for efficient DNN deployments while maintaining high flexibility.

This work has introduced a novel arithmetic architecture supporting adjustable precision BS-INT computation, ranging from high to low precision quantizations at runtime. It uses Software-defined SIMD to flexibly partition mantissa subwords in fine-grain with the required precision and efficiently support vector addition/subtraction/shift and vector by scalar multiplication among 3 to 24 bits. A minimal data pack unit allows for smooth conversion between different precision levels. Hence, it can perform the BS-INT operand accumulation by aligning the exponents through appropriate right shifts and repacking them into a larger mantissa to prevent overflow, eliminating the need for an expensive FP32 accumulator. Synthesis results indicate that our method can achieve a

reduction of up to 92%, 86%, and 53% per MAC area compared to the baseline FP32, bfloat16, and INT8 formats respectively. Furthermore, the experimental results have demonstrated that for BS-INT quantized benchmarks, a boost of up to 7.7x, 5.5x, and 1.5x in TOPS/W can also be obtained over the baselines, while still achieving comparable accuracy. While we have demonstrated improved performances specifically for DNN inferences, the same method is also broadly applicable for general data-parallel loop kernel workloads. These can potentially be flexibly and efficiently targeted due to the high programmability of the instruction-set processor.

The next chapter concludes this dissertation while summarizing the major contributions and highlighting the potential future work that can be performed for further improvement.

Chapter 9

Conclusion and future work

A diverse exploration on two broad disciplines of algorithmic optimization of DNNs and improving DNN deployment efficiency has been performed in this thesis. This is also highlighted via the contributions summary figure from Chapter 1 repeated here again in Fig. 9.1. The target applications are workloads that demand **data-efficient deep learning** (DL) that perform well even with limited data, across different modalities such as image classification and continuous control tasks. The target HW processors to run these workloads are programmable processors with **vector instruction-sets**, varying from **general-purpose** to **domain-specific** instructions for data-parallel workloads. The focus is on programmable architectures and not fully customized accelerators due to avoid high non-recurring engineering costs and reduce the risk of obsolescence to fast-paced algorithm evolution [24, 156]. The integration of these two objectives while handling limited resource constraints at the edge is performed through the technique of **block-quantization** forming the overarching goal of this thesis. The research towards data- and resource-efficient AI is especially useful in this era of high reliance on compute and dataset scaling, and where the privacy of user data is of utmost priority.

Chapter 2 introduces background DL concepts including commonly used operations during DNN inference and training processes. **Chapter 3** details some of the common algorithmic DNN optimizations typically performed for efficiency, stressing on the recently trending block-quantization or microscaling (MX) formats involving fine blocks each with a common power-of-2 scale [160]. Additionally, we emphasize that a complete optimization requires an overall HW-SW co-design with target HW consideration.

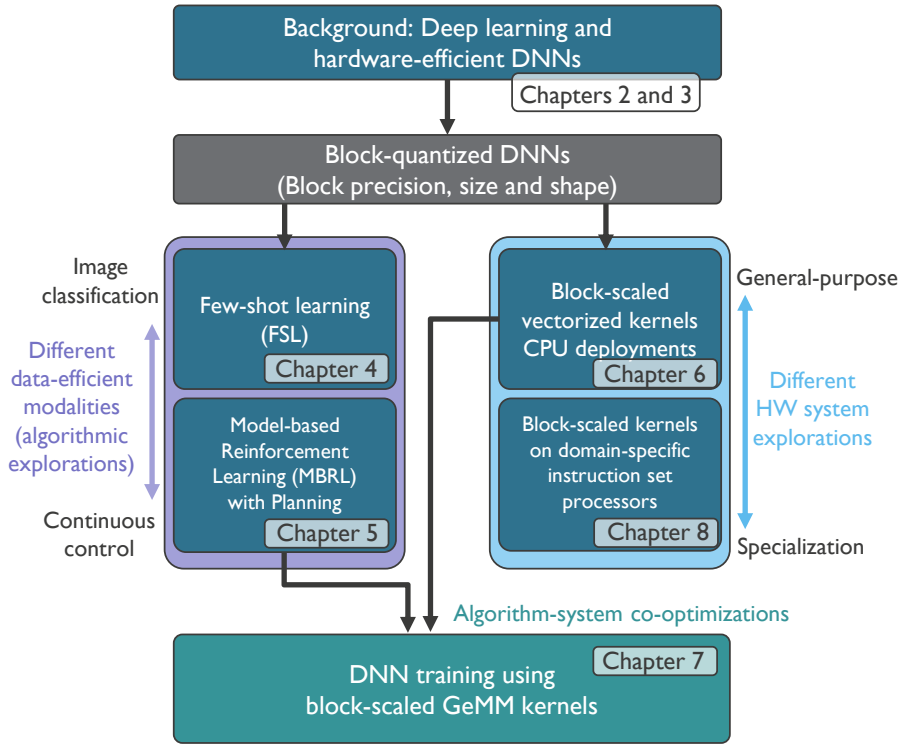


Figure 9.1: Overview of thesis contributions.

The remaining chapters of the thesis, discuss in detail the contributions of this work, as summarized in Fig. 9.1, and Section 9.1.

9.1 Thesis contributions

Q1. How to efficiently deploy data-efficient deep learning algorithms on resource-constrained hardware?

Chapter 4 focused on the *image classification modality* in scenarios where very limited training samples are available (1-5 samples per class). While few-shot learning (FSL) techniques such as the Reptile meta-learning (Learn-to-learn) efficiently handle this training data constraint, they do not effectively address the limited computational resources in resource-constrained edge devices. Efficient

adaptation on edge devices can be enabled through quantized training (Learn-on-chip) with block floating point (BFP) datatypes, without any assumption on the training data. In order to satisfy both constraints of training data and computational resources, we proposed a quantized meta-learning (Learn-to-learn-on-chip) process which upfront includes the target on-chip training process during meta-learning [141]. This enables us to find model initializations that more effectively transfer for quantized on-chip training. We analyze the accuracy impacts of our technique for on-chip training with 4-bit BFP GeMM operands throughout all training parameters (WAGE) and varied few-shot configurations (1/5-shots with 5/20-classes) of the Omniglot dataset. Our initializations are found to converge up to 90% faster and reduce accuracy loss by up to a factor of 1.3x, compared to conventional Reptile initializations.

Chapter 5 also addresses the same research question, however, for the target of *continuous control tasks* commonly requiring Reinforcement Learning (RL) algorithms. Data-efficiency in such tasks demands a low number of environmental interactions of the RL agent to eventually enable the desired control. Model-based RL (MBRL) algorithms are known to perform well for data-efficient control, however, do not consider any target HW constraints. To again address such a problem, we extend one of the prominent MBRL algorithms – PETS [38], which utilizes DNNs to learn state transitions that are then used for continuous control via model predictive control (MPC). We introduced block-quantizations via BFP datatypes for the DNN GeMM operands, exploring across different block sizes [195]. Model prediction errors in general are known to severely degrade control performance due to the phenomenon of compounding error with sequential DNN inferences, hence needing even more care during quantization. We evaluated the control performances for 4 standard simulated control tasks while comparing to the FP32 baseline counterparts. The control quality was also analyzed for different activation values, with ReLU being our priority due to more HW-efficient deployments when compared to baseline swish activation functions. Based on the design space explorations, it was evaluated that there was a need for higher precisions (16-bit BFP) at the inputs and outputs of the DNNs which are the states and actions from the continuous parameter space, to avoid severe performance degradation. For the rest of the network's weight and activation values, 8-bit BFP datatypes with vector blocks are found to be sufficient to reach comparable baseline FP32 performances, which still provides deployment-efficiency benefits when compared to using conventional complete FP32 models while also enabling up to $\sim$3.78$\times$ reduced memory footprint.

Q2. How do we efficiently deploy DNNs with block-scaled formats using general-purpose vector instructions?

Chapter 6 explores this research question through GeMM kernel design, predominantly used in DNN inference and training, on ARM SVE-enabled processors with general-purpose vector instructions. While standard datatypes such as FP32 and INT8 are more easily deployed using such vector extensions, special datatypes such as block-scaled formats require customized kernel design. Hence, to maintain high-vectorization and flexible deployments, we design block-scaled integer (BS-INT8) GeMM microkernels using ARM SVE intrinsics in a vector-length agnostic (VLA) programming model [194]. This enables a quick and wide design space exploration potential for algorithmic BS-INT8 design parameters and HW design parameters such as SIMD vector lengths.

We compare the system-level performance of these BS-INT8 kernels with baseline datatypes, along with design parameters such as specific MAC instructions used, block sizes, vector lengths through instruction and cycle counts obtained with gem5 simulations. The inter-block accumulations of different blocks are performed without conversion to FP32, and instead via soft-float like emulation with left and right integer bit shifts. We also optimize the kernels by proposing a 2D block shape for the left matrix operand in a GeMM which increases the performance compared to conventional 1D blocks due to reduced exponent handling overheads for the BS-INT8 computations. These exponent overheads are also reflected in block size variation for the same GeMM workload, where larger blocks require lower BS-INT8 partial block accumulation alignments and hence have higher performance than smaller blocks. Note that the design choices of block size, block shape, and vector lengths can be run-time configured, without requiring the need for recompilation.

We additionally designed the kernels to handle different requantization strategies for the BS-INT8 quantizations – static and dynamic, which are performed in a fused-manner that reduces the significant data transfer overheads to memory in DNN workloads. The static version requires calibration to find scales increasing deployment ease, but reduces the runtime overheads of GeMM computations. Dynamic requantization performs runtime calibration of scales, but in turn requires more cycles for online calibration based on calculating the maximum value within a block.

We also evaluate the impacts of these kernel design choices for running the GeMMs of a ResNet20 CNN architecture in gem5 with 256-bit vector length configuration. The static and dynamic BS-INT8 with 2D block configurations yielded on average 3.8x and 2.9x speedups over FP32 models respectively.

Chapter 7 further extends the previous explorations answering the same research question, however for GeMMs in a DNN training workload. We use the same training workloads as the DNNs used in the MBRL process in chapter 5. We also identify potential algorithm-system co-optimizations that can be performed due to the usage of BS-INT during training. While other design options previously described are feasible, we limit our explorations in this chapter to dynamic requantization on 256-bits vector length target HW. The focus is instead on the shape of the blocks being 1D vector blocks or square (specific case of 2D) blocks. This is crucial in a DNN training process, as the square block enables the commutative property of transpose-quantization operator. This in turn leads to fewer quantization operators and enables storing intermediate quantized activations for the backward pass leading to reduced memory footprint. Additionally, drawing the conclusions from chapter 6, coarse 2D blocks have higher speedups than fine 1D blocks for BS-INT8 GeMM computations. However, the coarse blocks also exhibit higher quantization error, the impact of which is analyzed through simulated quantizations for different DNN training runs of the MBRL workload.

We now have BS-INT8 square blocks that are more HW-efficient, but more loss-prone leading to coarse-commutative training. Conversely, BS-INT8 vector blocks are HW-inefficient, but more accurate providing fine-adaptation of DNNs. We therefore propose an adaptive approach that aims at balancing the trade-offs of the above 2 block configurations, by first performing square block coarse-commutative training efficiently on HW and then switching to vector fine-adaptation to recover the algorithmic performance [140]. Benchmarking on the corresponding FC NNs in the MBRL workloads, our adaptive training approach can yield around 1.95× faster training GeMM computations with up to 2× lower memory footprint, compared to conventional FP32 models while reaching similar training performances.

Q3. How do we increase efficiency for block-scaled DNN deployments by moving towards domain-specific instruction set processors?

In **Chapter 8**, our objective is to answer the above research question by exploring the block-scaled integer DNN deployments on a specific HW design called SoftSIMD [242]. The SoftSIMD microarchitecture is an energy-, area-efficient HW implementation that can flexibly operate on variable-precision operands (like 3-24 bits) at run-time. This is especially useful in data-parallel workloads such as DNNs that can effectively exploit varying degrees of data-level parallelism as a function of bit-precision required to maintain similar algorithmic performance. SoftSIMD also enables to progressively increase the accumulation bit width avoiding the possibility of overflow through its efficient

and seamless conversion across different precision formats with a data packing unit. BS-INT computations are performed in a similar manner as we had proposed in chapter 6 via float-like emulations with integer shifts. However, in our earlier implementation, we were restricted to INT8 operands due to the lack of instruction support for sub-byte computations, while SoftSIMD HW can support more diverse deployments providing further co-optimization potential.

The HW design for BS-INT computations across various precision levels from 8-bits to 3-bits reduces the area per MAC by 78%–92% compared to FP32, 60%–86% compared to bfloat16 baseline implementations respectively. Compared to INT8, our 8-bits precision configuration incurs slight area overhead while at 3-bits we observe 53% reduction. We also performed experiments to analyze the impacts on CNN accuracies with diverse BS-INT quantization configurations: varied block shapes and sizes, different mixed-precision explorations of weight and activation bit-widths. All together, the BS-INT optimized deployment yielded a boost of up to 7.7x, 5.5x, and 1.5x improvements in energy-efficiency (TOPS/W) over the baseline FP32, bfloat16, INT8 deployments, while not incurring much accuracy loss compared to baseline FP32 accuracy ($< 1\%$ drop).

9.2 Future work suggestions

This section proposes suggestions for future work, building upon the contributions of this thesis.

Supervised few-shot learning

- While the inclusion of target HW constraints does improve the meta-learning transfer as shown in chapter 4, the computational overhead of performing meta-learning also increases. This overhead might be more severe for more challenging tasks leading to difficult tuning process. A method to solve this could be to perform HW-aware meta-learning only for few of the parameters that are most critical to the learning process, identified through contribution analysis, while fine-tuning process in the field is still performed for all parameters. Another option could be to extend the HW-aware concept to existing techniques such as Meta-Transfer Learning [205], ANIL [178], BOIL [158] which during meta-learning also consider that specific parameters at fine-tuning would be frozen. Extension of our approach towards hybrid meta-learning and metric-learning [222] and continual learning [165] could also be other interesting directions.

- While currently we have mainly explored BFP datatypes for more efficient adaptations on-device, additional explorations with other formats, especially with the recently trending microscaling formats (MX) [160] and different block configurations could further generalize the benefits of our HW-aware meta-learning approach.
- Evaluation of the impact of our algorithm on workloads run on actual HW through on-device latency and power measurements could better quantify the practical benefits of our approach.
- Recent models such as transformers in Large Language Models have been shown to be few-shot learners [19], however, these architectures are intended for handling general tasks. Furthermore, edge deployments of such large models is a difficult task and needs explorations if they can be justified to target more complex applications.
- In accordance with increasing demands for intelligence in our everyday devices, explorations on real-world applications such as activity recognition or keyword spotting can provide practical benefits that are enabled by our algorithm [70]. Futuristic applications such as novel view synthesis or 3D reconstruction with only few-shot 2D image snapshots [212] could also be an interesting demonstration experiment, that can be further enhanced by more efficient on-device training through our algorithm.

Continuous control tasks with sample-efficient algorithms

- A relatively simple but time-consuming extension of the works in chapter 5 is to perform detailed hyperparameter optimization with DNN architecture design parameters, block-quantized or MX format configuration parameters, which could provide more co-optimized deployment opportunities leading to higher throughput and/or lower power consumption.
- The design of domain-specific HW to tackle the challenging computational demands for complex continuous control tasks with MBRL algorithms, while also satisfying the edge device resource constraints is a very attractive opportunity. The potential acceleration options for the planning process could be the design of high throughput MAC units for fast DNN inferences and a random action sequence generator for efficient CEM optimization. This is especially beneficial in this era, where autonomous decision-making in real-time at the edge with satisfactory performance is a highly sought-after demand. The predominant focus of the MBRL community is towards developing algorithms for good control performance, providing several opportunities for integration with the HW-efficient DL community.

- Algorithmic optimizations for dynamics model predictions could further be utilized in such a quantization exploration study for HW-efficient control performance. One option is to use meta-learning dynamics models [13, 39] potentially in combination with our proposed algorithm in chapter 4 for HW-aware deployments with quantized DNNs. Another option could be to use quantized multistep dynamics models [114] which could reduce the effects of compounding error alleviating the constraints of quantization enabling more efficient deployments. Exploring different activation functions and its underlying hardware acceleration potential is another co-design approach that seems promising, as is done by [129] through dynamics model sparsification to achieve more efficient planning.
- While we mainly explored block-quantized DNNs as dynamics model in an MBRL with pure MPC planning approach, MFRL algorithms with policy/value-based DNN models can also exhibit performance improvements through block-quantizations. Additional algorithmic optimization such as hybrid MBRL-MFRL algorithms could also be targeted to trade-off sample-efficiency and deployment-efficiencies. The dynamics model that learns the state transitions through interactions in an environment is not only useful for predicting the future trajectories and deciding optimal control, but can also be used in the following ways: to improve the policy of an MFRL algorithm while also utilizing the sample-efficiency of MBRL algorithms [27, 95, 110, 143]; to provide a digital copy enabling simulation of real-world interactions [235].
- An application demonstration with actual robotic control, demonstrating the sample-efficient and HW-efficient deployments showcasing the real-time control requirements with block-quantized MBRL algorithms in comparison to more conventional algorithms could be a challenging but very impactful and interesting extension direction.

Vectorized deployments on instruction-programmable processors

- Designing an automatic GeMM microkernel generator [2] in the SW programming model based on design parameters could further enhance the design space exploration possibilities of our method in chapter 6. Such a generator in combination with the design of non-fused dynamic requantization strategies would enable explorations of even more flexible block configurations – 1D or 2D with less dependence on HW vector lengths. An improved accumulation strategy for block-scaled kernels while still utilizing FP units could be another potential microkernel design extension, especially useful for workloads demanding it.

- Integration of our block-scaled kernels in DNN deployment frameworks like AIfES [233] and OneDNN [159] can enable diverse and full workload deployment explorations with different configurations in a modular manner.
- While we demonstrated system-level explorations of deploying our block-scaled kernels, extra profiling such as power consumption analysis would provide a more holistic view of our deployments also enabling better co-optimization studies. An exploration with different system parameters such as cache sizes, number of registers, pipeline execution model, etc. are additional design parameters that could be analyzed for optimized deployments. Actual workload profiling on real HW that is already available as tabulated in Table 3.3 (eg. Single core experiments on AWS graviton 3.) and comparison with simulated system metrics could be a nice validation study extension.
- We only explored the ARM SVE ISA for a VLA programming model to deploy block-scaled GeMM kernels, however RVV ISA deployments are also a feasible option. This would also enable more detailed HW-SW co-design strategies due to the open-source nature of RISC-V architecture and a pure vector style [189] execution processor with RVV enabling more flexible runtime deployments independent of HW vector lengths [171]. Proposing custom instruction for domain-specific instruction-set extension while also profiling the extra HW overhead specifically for block-scaled or MX formats is very useful, especially with the availability and development of such MAC execution units [191]. The usage of an adaptable register file [115] enabling runtime modification of HW vector length, is another system-design possibility that could be exploited well for block-scaled deployments.
- Further deployment studies on SoftSIMD HW than currently explored in chapter 8 with heterogeneous quantization for DNNs is another optimization opportunity for vectorized kernel deployments while also utilizing the HW flexibility capabilities to its full potential.
- Lastly, we mainly focused on single core experiments, but our kernels would also be applicable in a multi-core setup with either homogeneous or heterogeneous cores.

Adaptive DNN training at the edge

- In chapter 7, we mainly evaluated the performance of our training algorithm with block-scaled GeMM kernels on FCNNs, however, the

method is generally applicable and beneficial to any GeMM-based DNN model architecture such as CNNs, Transformers, etc. In addition, evaluating the impact on practical applications would make the contribution more valuable to the community.

- The adaptive nature of our algorithm's block choices included an S8 square block sharing exponent among 64 elements and V8 vector blocks with exponents shared by 8 elements. This can be further explored among different block configurations, while still in an adaptive training fashion where the trade-off among HW-efficient deployments and SW training convergence quality is balanced effectively. The adaptive switching across different configurations could also be designed to be handled at runtime instead of through a hyperparameter as currently explored based on strategies such as the number of underflows during quantization [118, 246] or at specific loss saturation points observed during convergence. The method can also be extended for sparse GeMMs [139], while ensuring similar non-zero element indices are selected during the forward and backward passes of the training process.

- Recent developments in the Large Language Model (LLM) fine-tuning explorations have also utilized similar square and vector blocks during training which is potentially a nice workload / application to evaluate our adaptive approach [126]. The RL-based fine-tuning through human feedback approaches [163], could also benefit from our techniques specifically for optimizations of the GeMM operations involved in these workloads.

- Optimizing the training process for non-GeMM operations such as batchnorm [9] or optimizer updates [49, 249] with effective HW design for block-scaled kernels could be useful for further accelerating any generic training workload.

- Distributed training on multi-core processors for more complex training workloads while still utilizing block-scaled kernels in combination with an effective instruction-programmable processor design could lead to a nice co-design study.

- Lastly, similar block-quantized deployments can be explored alongside non-backpropagation training algorithm research that target brain-like learning. This is an active area of research with algorithms trying to overcome the bottlenecks of backpropagation such as weight-transport [62, 86, 121, 154], update-locking [62, 134], etc. Analyzing the impacts of memory transfers during training runs, and explorations based on loop nest optimizations could reveal additional benefits of such non-backpropagation algorithms.

As DL continues to evolve, the pursuit of data-efficient and hardware-efficient solutions will remain crucial in enabling ubiquitous smart intelligence. This thesis contributes a small step towards reaching this goal with explorations across the algorithm-system stack targeting diverse task modalities. Looking ahead, the growing synergy between algorithmic innovations and hardware advancements promises a future where DL is not only more powerful but also secure, sustainable, and transformative for a broader range of applications. The journey is far from over, and the coming years are sure to bring even more groundbreaking developments in this dynamic and ever-expanding field.

Appendix A

Hyperparameter settings for few-shot learning experiments

In this appendix, we provide details on hyperparameter settings used in our few-shot learning (FSL) experiments of chapter 4 for performing conventional Reptile meta-learning (L2L), our modified HW-aware meta-learning (qL2L), as well as quantized fine-tuning processes.

Table A.1 tabulates the various hyperparameter settings used to run the meta-learning scenarios discussed in Section 4.4.

Table A.1: Hyperparameter settings for Reptile meta-learning [152] (L2L) and quantized meta-learning (qL2L). f represents 32-bit floating point numbers, while others represent BFP mantissa-bitwidths.

Parameter		5-ways		20-ways	
		L2L	qL2L	L2L	qL2L
Outer loop	Outer step size	1.0	0.01	1.0	0.01
	Outer iterations	100,000	10,000	200,000	10,000
	Meta-batch size	5	5	5	5
Inner loop	Training shots	10	10	10	10
	Learning rate	0.001	0.001	0.0005	0.0005
	Inner batch size	10	10	20	20
	Inner iterations	5	100	10	100
	WAGE quantization	ffff	4444	ffff	4444

Table A.2 details the fine-tuning setups of the conventional FP and quantized adaptation used in all experiments.

Table A.2: Hyperparameter settings for conventional FP32 and BFP quantized fine-tuning

Parameter		5-ways (20-ways)	
		FP32	Quantized
Fine-tuning	WAGE quantization	ffff	4444/6666/8888
	Adam lr. rate	0.001 (0.0005)	0.001 (0.0005)
	Max. Iterations	300	300
	Batch size	shots×5 (20)	shots×5 (20)
	Test query shots	15	15
	Number of test tasks	200	200
	Non-transductive	Yes	Yes

Appendix B

Block-scaled DNNs

In this appendix, we provide additional details for the BS-INT8 computational kernels designed in chapter 6 – accumulation and requantization processes.

Temporal accumulations for BS-INT8

Fig. B.1 depicts the equations for temporally accumulating the partial products of BS-INT8 datatypes either with or without conversions to FP32 equivalents. For the scenario without FP32 conversion (without floating point units (FPU)), different colors in the equation represent accumulations within a block with common exponents. The multiplication with power of 2, or simple bit shifts on HW, indicate the alignment operation for different exponents in the partial product temporal accumulations. Note that these partial product accumulations might happen in different instruction counts depending on the ratio of block size and the MAC accumulation step k_r specified in Table 6.4.

Requantizations for BS-INT8

We specify the equations for the 2 modes of requantization we considered in this work. Each intermediate GeMM output would have an integer part and its own exponent. The requantization process essentially involves modifying the outputs to have a common exponent and normalize the rest accordingly. Static and dynamic requantization mainly differ in how this common exponent is evaluated:

Static: Common exponents of output blocks are pre-determined.

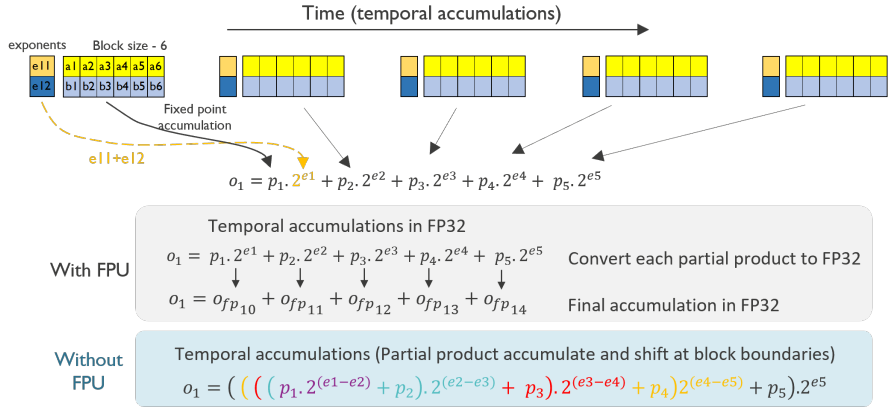


Figure B.1: Temporal accumulation choices for BS-INT8 datatypes via: 1) FP32 accumulations with FPUs. 2) Integer partial product accumulations and shifts based on common exponent differences for alignments, without FPUs. Our BS-INT8 accumulations are similar to software emulated float additions, however, the alignments need to happen only after every block sized accumulations.

Dynamic: Common exponents of output blocks are calculated during runtime as the maximum exponent of the absolute value of the numbers in the block.

With these common exponents, rest of the integer values are normalized accordingly based on the exponent differences and quantized back to 8-bits. Eqs. (B.1)-(B.6) demonstrate this requantization process of a block size B of GeMM outputs x_o, where each element has its own exponent e_i and integer part m_i. The block-scaled quantized version would have a single common exponent for the block e_c and separate adjusted integers m_i''.

$$\text{Original outputs: } x_o = [2^{e_0} m_0, 2^{e_1} m_1, \ldots, 2^{e_{B-1}} m_{B-1}] \tag{B.1}$$

$$m_i \text{ is a 8 / 32-bit integer, } e_i \text{ are 8-bit exponents} \tag{B.2}$$

$$\text{Static requantization: } e_c \text{ is pre-determined (calibration)} \tag{B.3}$$

$$\text{Dynamic requantization: } e_c \text{ is } \max(e_0, e_1, e_2, ...e_{B-1}) \tag{B.4}$$

$$\text{Normalized outputs: } x_{norm} = 2^{e_c} \left[m'_0, m'_1, \ldots, m'_{B-1}\right] \tag{B.5}$$

$$m'_i = m_i / 2^{e_c - e_i}$$

$$\text{Block-scaled outputs: } x_{BS-INT8} = 2^{e_c} \left[m''_0, m''_1, \ldots, m''_{B-1}\right] \tag{B.6}$$

$$m''_i = \text{truncate}(m'_i)$$

Bibliography

[1] I. Advanced Micro Devices. *AMD Programmer's Manual: XOP, FMA4, and CVT16 Instruction Set Extensions*. Tech. rep. 43479. Archived technical documentation. Advanced Micro Devices, Inc., 2011. URL: `https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/programmer-references/43479.pdf` (Cited on page 9).

[2] G. Alaejos, A. Castelló, H. Martínez, P. Alonso-Jordá, F. D. Igual, and E. S. Quintana-Ortí. "Micro-kernels for portable and efficient matrix multiplication in deep learning". In: *The Journal of Supercomputing* 79.7 (2023), pp. 8124–8147 (Cited on pages: 83, 86, 145).

[3] G. Alsuhli, V. Sakellariou, H. Saleh, M. Al-Qutayri, B. Mohammad, and T. Stouraitis. *Number Systems for Deep Neural Network Architectures*. Springer, 2023 (Cited on pages: 7, 30).

[4] M. Alwani, H. Chen, M. Ferdman, and P. Milder. "Fused-layer CNN accelerators". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12 (Cited on page 83).

[5] *ARMIE ARM Instruction Emulator 22.0*. `https://developer.arm.com/documentation/102190/22-0/Introduction`. Accessed: 2024-01-15. 2022 (Cited on page 94).

[6] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38 (Cited on pages: 8, 59).

[7] M. AskariHemmat, T. Dupuis, Y. Fournier, N. El Zarif, M. Cavalcante, M. Perotti, F. Gürkaynak, L. Benini, F. Leduc-Primeau, Y. Savaria, et al. "Quark: An integer RISC-V vector processor for sub-byte quantized DNN inference". In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2023, pp. 1–5 (Cited on pages: 83, 103).

[8] R. H. W. Q. O. Azizi, A. S. B. C. L. Christos, and K. M. Horowitz. "Retrospective: Understanding Sources of Inefficiency in General-purpose Chips". In: *ISCA@50 25-Year Retrospective: 1996-2020*. Ed. by J. F. Martínez and L. K. John. ACM SIGARCH and IEEE TCCA, 2023. URL: https://bpb-us-w2.wpmucdn.com/sites.coecis.cornell.edu/dist/7/587/files/2023/06/Hameed_2010_-Understanding.pdf (Cited on page 6).

[9] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. "Scalable methods for 8-bit training of neural networks". In: *Advances in neural information processing systems* 31 (2018) (Cited on pages: 27, 147).

[10] S. Barrachina, A. Castelló, M. F. Dolz, T. M. Low, H. Martínez, E. S. Quintana-Ortí, U. Sridhar, and A. E. Tomás. "Reformulating the direct convolution for high-performance deep learning inference on ARM processors". In: *Journal of Systems Architecture* 135 (2023), p. 102806 (Cited on page 83).

[11] B. R. Bartoldson, B. Kailkhura, and D. Blalock. "Compute-efficient deep learning: Algorithmic trends and opportunities". In: *Journal of Machine Learning Research* 24.122 (2023), pp. 1–77 (Cited on pages: 5, 25, 26).

[12] H. Benmeziane, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang. "A comprehensive survey on hardware-aware neural architecture search". In: *arXiv preprint arXiv:2101.09336* (2021) (Cited on page 27).

[13] A. Bhardwaj, J. Rothfuss, B. Sukhija, Y. As, M. Hutter, S. Coros, and A. Krause. "Data-Efficient Task Generalization via Probabilistic Model-based Meta Reinforcement Learning". In: *IEEE Robotics and Automation Letters* (2024) (Cited on pages: 58, 145).

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7 (Cited on page 94).

[15] J. Björck, X. Chen, C. De Sa, C. P. Gomes, and K. Weinberger. "Low-precision reinforcement learning: running soft actor-critic in half precision". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 980–991 (Cited on pages: 8, 63).

[16] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151 (Cited on page 83).

[17] Z. I. Botev, D. P. Kroese, R. Y. Rubinstein, and P. L'Ecuyer. "The cross-entropy method for optimization". In: *Handbook of statistics*. Vol. 31. Elsevier, 2013, pp. 35–59 (Cited on page 61).

[18] J. Bronskill, J. Gordon, J. Requeima, S. Nowozin, and R. Turner. "Tasknorm: Rethinking batch normalization for meta-learning". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 1153–1164 (Cited on page 43).

[19] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901 (Cited on pages: 1, 144).

[20] B. de Bruin, Z. Zivkovic, and H. Corporaal. "Quantization of deep neural networks for accumulator-constrained processors". In: *Microprocessors and microsystems* 72 (2020), p. 102872 (Cited on pages: 82, 83, 109).

[21] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. "Once-for-all: Train one network and specialize it for efficient deployment". In: *arXiv preprint arXiv:1908.09791* (2019) (Cited on page 27).

[22] H. Cai, C. Gan, L. Zhu, and S. Han. "Tinytl: Reduce memory, not parameters for efficient on-device learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 11285–11297 (Cited on pages: 40, 44, 55, 108).

[23] H. Cai, L. Zhu, and S. Han. "Proxylessnas: Direct neural architecture search on target task and hardware". In: *arXiv preprint arXiv:1812.00332* (2018) (Cited on page 23).

[24] F. Catthoor, P. Raghavan, A. Lambrechts, M. Jayapala, A. Kritikakou, and J. Absar. *Ultra-low energy domain-specific instruction-set processors*. Springer Science & Business Media, 2010 (Cited on pages: 6, 35, 125, 138).

[25] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini. "Spatz: A compact vector processing unit for high-performance and energy-efficient shared-L1 clusters". In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 2022, pp. 1–9 (Cited on pages: 10, 37).

[26] ChatGPT. *ChatGPT (October 2024 version)*. OpenAI, Available at: `https://chatgpt.com/`. 2024 (Cited on page 2).

[27] Y. Chebotar, K. Hausman, M. Zhang, G. Sukhatme, S. Schaal, and S. Levine. "Combining model-based and model-free updates for trajectory-centric reinforcement learning". In: *International conference on machine learning*. PMLR. 2017, pp. 703–711 (Cited on page 145).

[28] K. Chellapilla, S. Puri, and P. Simard. "High performance convolutional neural networks for document processing". In: *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft. 2006 (Cited on page 130).

[29] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357 (Cited on page 76).

[30] J. Chen and X. Ran. "Deep learning with edge computing: A review". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674 (Cited on page 4).

[31] W.-Y. Chen, Y.-C. Liu, Z. Kira, Y.-C. F. Wang, and J.-B. Huang. "A closer look at few-shot classification". In: *arXiv preprint arXiv:1904.04232* (2019) (Cited on page 45).

[32] P. S. Chib and P. Singh. "Recent advancements in end-to-end autonomous driving using deep learning: A survey". In: *IEEE Transactions on Intelligent Vehicles* (2023) (Cited on page 3).

[33] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan. "Pact: Parameterized clipping activation for quantized neural networks". In: *arXiv preprint arXiv:1805.06085* (2018) (Cited on page 32).

[34] F. Chollet. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258 (Cited on pages: 5, 22).

[35] J. Choquette, O. Giroux, and D. Foley. "Volta: Performance and programmability". In: *Ieee Micro* 38.2 (2018), pp. 42–52 (Cited on page 6).

[36] Y. Choukroun, E. Kravchik, F. Yang, and P. Kisilev. "Low-bit quantization of neural networks for efficient inference". In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019, pp. 3009–3018 (Cited on page 31).

[37] X. Chu, B. Zhang, and R. Xu. "Multi-objective reinforced evolution in mobile neural architecture search". In: *european conference on computer vision*. Springer. 2020, pp. 99–113 (Cited on page 27).

[38] K. Chua, R. Calandra, R. McAllister, and S. Levine. "Deep reinforcement learning in a handful of trials using probabilistic dynamics models". In: *Advances in neural information processing systems* 31 (2018) (Cited on pages: 8, 57, 59, 60, 61, 62, 65, 73, 107, 116, 140).

[39] I. Clavera, J. Rothfuss, J. Schulman, Y. Fujita, T. Asfour, and P. Abbeel. "Model-based reinforcement learning via meta-policy optimization". In: *Conference on Robot Learning*. PMLR. 2018, pp. 617–629 (Cited on page 145).

[40] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara. "Vectorizing posit operations on RISC-V for faster deep neural networks: experiments and comparison with ARM SVE". In: *Neural Computing and Applications* 33 (2021), pp. 10575–10585 (Cited on page 80).

[41] I. Corporation. *Intel AVX-512 Overview*. Accessed: 2024-10-29. Intel Corporation. 2024. URL: `https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html` (Cited on page 9).

[42] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314 (Cited on page 22).

[43] S. Dai, R. Venkatesan, M. Ren, B. Zimmer, W. Dally, and B. Khailany. "Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference". In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 873–884 (Cited on pages: 9, 31, 33, 63, 75, 82, 83, 123).

[44] W. J. Dally, Y. Turakhia, and S. Han. "Domain-specific hardware accelerators". In: *Communications of the ACM* 63.7 (2020), pp. 48–57 (Cited on page 6).

[45] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, et al. "Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point". In: *Advances in neural information processing systems* 33 (2020), pp. 10271–10281 (Cited on pages: 9, 33, 47, 60, 63, 68, 69, 70, 75, 82, 83, 87, 107, 123, 124, 130).

[46] B. Darvish Rouhani, R. Zhao, V. Elango, R. Shafipour, M. Hall, M. Mesmakhosroshahi, A. More, L. Melnick, M. Golub, G. Varatkar, et al. "With shared microexponents, a little shifting goes a long way". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 2023, pp. 1–13 (Cited on pages: 7, 33).

[47] B. Darvish Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf, et al. "Microscaling Data Formats for Deep Learning". In: *arXiv e-prints* (2023), arXiv–2310 (Cited on pages: 7, 33, 77, 80, 81, 83, 91, 106, 109).

[48] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, et al. "Mixed precision training of convolutional neural networks using integer operations". In: *arXiv preprint arXiv:1802.00930* (2018) (Cited on pages: 82, 83, 109).

[49] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer. “8-bit optimizers via block-wise quantization”. In: *arXiv preprint arXiv:2110.02861* (2021) (Cited on pages: 21, 147).

[50] A. Dosovitskiy. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020) (Cited on page 1).

[51] M. Drumond, T. Lin, M. Jaggi, and B. Falsafi. “Training dnns with hybrid block floating point”. In: *Advances in Neural Information Processing Systems* 31 (2018) (Cited on pages: 7, 82, 83, 91, 103).

[52] T. Edamatsu and D. Takahashi. “Efficient Large Integer Multiplication with Arm SVE Instructions”. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 2023, pp. 9–17 (Cited on page 80).

[53] T. Elsken, J. H. Metzen, and F. Hutter. “Neural architecture search: A survey”. In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21 (Cited on pages: 22, 27).

[54] S. Engineering. *AI Chip Architectures: The Race to the Edge*. Accessed: 2024-10-28. 2024. URL: `https://semiengineering.com/ai-chip-architectures-race-to-the-edge/` (Cited on page 3).

[55] Epoch AI. *Data on Notable AI Models*. Accessed: 2024-10-25. 2024. URL: `https://epochai.org/data/notable-ai-models` (Cited on page 3).

[56] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha. “Learned step size quantization”. In: *arXiv preprint arXiv:1902.08153* (2019) (Cited on page 32).

[57] H. Fan, G. Wang, M. Ferianc, X. Niu, and W. Luk. “Static block floating-point quantization for convolutional neural networks on FPGA”. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2019, pp. 28–35 (Cited on page 91).

[58] C. Finn, P. Abbeel, and S. Levine. “Model-agnostic meta-learning for fast adaptation of deep networks”. In: *International conference on machine learning*. PMLR. 2017, pp. 1126–1135 (Cited on pages: 7, 40, 42, 47, 55).

[59] R.-V. Foundation. *RISC-V Vector Workshop*. Workshop presentation on RISC-V vector extensions. RISC-V Foundation. 2015. URL: `https://riscv.org/wp-content/uploads/2015/06/riscv-vector-workshop-june2015.pdf` (Cited on page 9).

[60] S. Fox, S. Rasoulinezhad, J. Faraone, david boland, and P. Leong. “A Block Minifloat Representation for Training Deep Neural Networks”. In: *International Conference on Learning Representations*. 2021. URL: `https://openreview.net/forum?id=6zaTwpNSsQ2` (Cited on pages: 7, 30, 40, 75, 82).

[61] J. Frankle, D. J. Schwab, and A. S. Morcos. “The early phase of neural network training”. In: *arXiv preprint arXiv:2002.10365* (2020) (Cited on page 8).

[62] C. Frenkel, M. Lefebvre, and D. Bol. “Learning without feedback: Fixed random learning signals allow for feedforward training of deep neural networks”. In: *Frontiers in neuroscience* 15 (2021), p. 629892 (Cited on pages: 7, 25, 26, 147).

[63] H. Gharoun, F. Momenifar, F. Chen, and A. Gandomi. “Meta-learning approaches for few-shot learning: A survey of recent advances”. In: *ACM Computing Surveys* 56.12 (2024), pp. 1–41 (Cited on pages: 5, 7).

[64] H. Gholamalinezhad and H. Khosravi. “Pooling methods in deep neural networks, a review”. In: *arXiv preprint arXiv:2009.07485* (2020) (Cited on page 18).

[65] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021) (Cited on page 63).

[66] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. “A survey of quantization methods for efficient neural network inference”. In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326 (Cited on pages: 31, 35, 36).

[67] X. Glorot and Y. Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256 (Cited on page 21).

[68] X. Glorot, A. Bordes, and Y. Bengio. “Deep sparse rectifier neural networks”. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2011, pp. 315–323 (Cited on page 22).

[69] M. Goldblum, S. Reich, L. Fowl, R. Ni, V. Cherepanova, and T. Goldstein. “Unraveling meta-learning: Understanding feature representations for few-shot tasks”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 3607–3616 (Cited on page 40).

[70] T. Gong, Y. Kim, J. Shin, and S.-J. Lee. “Metasense: few-shot adaptation to untrained conditions in deep mobile sensing”. In: *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. 2019, pp. 110–123 (Cited on pages: 4, 8, 40, 144).

[71] Google. *Google Home*. Smart speaker with voice assistant, Available at: `https://store.google.com/product/google_home`, Accessed on: October 14, 2024. 2016 (Cited on page 2).

[72] A. Graves and A. Graves. “Long short-term memory”. In: *Supervised sequence labelling with recurrent neural networks* (2012), pp. 37–45 (Cited on page 23).

[73] L. Grossman and B. Plancher. “Just Round: Quantized Observation Spaces Enable Memory Efficient Learning of Dynamic Locomotion”. In: *arXiv preprint arXiv:2210.08065* (2022) (Cited on page 63).

[74] L. Grossman and B. Plancher. “Just round: Quantized observation spaces enable memory efficient learning of dynamic locomotion”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2023, pp. 3002–3007 (Cited on page 8).

[75] K. Guo, W. Li, K. Zhong, Z. Zhu, S. Zeng, S. Han, Y. Xie, P. Debacker, M. Verhelst, and Y. Wang. *Neural Network Accelerator Comparison*. `https://nicsefc.ee.tsinghua.edu.cn/projects/neuralnetwork-accelerator.html`. Accessed: 2024-11-07. 2020 (Cited on page 34).

[76] S. R. Gupta, N. Papadopoulou, and M. Pericas. “Accelerating CNN inference on long vector architectures via co-design”. In: *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2023, pp. 145–155 (Cited on pages: 76, 80).

[77] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. “Deep learning with limited numerical precision”. In: *International conference on machine learning*. PMLR. 2015, pp. 1737–1746 (Cited on page 47).

[78] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870 (Cited on page 63).

[79] D. Han and H.-J. Yoo. “An Overview of Energy-Efficient DNN Training Processors”. In: *On-Chip Training NPU-Algorithm, Architecture and SoC Design* (2023), pp. 183–210 (Cited on page 4).

[80] J. Han and C. Moraga. “The influence of the sigmoid function parameters on the speed of backpropagation learning”. In: *International workshop on artificial neural networks*. Springer. 1995, pp. 195–201 (Cited on page 18).

[81] S. Han, J. Pool, J. Tran, and W. Dally. “Learning both weights and connections for efficient neural network”. In: *Advances in neural information processing systems* 28 (2015) (Cited on page 28).

[82] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (Cited on pages: 22, 96, 131).

[83] K. He, X. Zhang, S. Ren, and J. Sun. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034 (Cited on page 21).

[84] S. Herculano-Houzel. “The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost”. In: *Proceedings of the National Academy of Sciences* 109.supplement_1 (2012), pp. 10661–10668 (Cited on page 15).

[85] G. Hinton. “A Practical Guide to Training Restricted Boltzmann Machines”. In: *Neural Networks: Tricks of the Trade/Springer* (2012) (Cited on page 21).

[86] G. Hinton. “The forward-forward algorithm: Some preliminary investigations”. In: *arXiv preprint arXiv:2212.13345* (2022) (Cited on pages: 7, 26, 147).

[87] M. 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)*. IEEE. 2014, pp. 10–14 (Cited on pages: 35, 36).

[88] A. G. Howard. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017) (Cited on page 23).

[89] Incrementors. *ChatGPT Stats: Comprehensive Guide on ChatGPT Database, Growth, Usage, and More*. Accessed: 2024-10-29. 2024. URL: https://www.incrementors.com/blog/chatgpt-stats/#:~:text=The%20database%20size%20of%20ChatGPT,web%2C%20books%2C%20Wikipedia (Cited on page 2).

[90] D. Ingram. *Sam Altman bets on nuclear power to fuel the A.I. boom*. Accessed: 2024-10-28. 2024. URL: https://www.nbcnews.com/tech/tech-news/nuclear-power-oklo-sam-altman-ai-energy-rcna139094 (Cited on page 2).

[91] S. Ioffe. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *arXiv preprint arXiv:1502.03167* (2015) (Cited on pages: 18, 27).

[92] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713 (Cited on page 27).

[93] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu. "Decoupled neural interfaces using synthetic gradients". In: *International conference on machine learning*. PMLR. 2017, pp. 1627–1635 (Cited on page 44).

[94] S. Jain, A. Gural, M. Wu, and C. Dick. "Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks". In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 112–128 (Cited on page 31).

[95] M. Janner, J. Fu, M. Zhang, and S. Levine. "When to trust your model: Model-based policy optimization". In: *Advances in neural information processing systems* 32 (2019) (Cited on pages: 8, 57, 59, 63, 145).

[96] W. Kahan. "IEEE standard 754 for binary floating-point arithmetic". In: *Lecture Notes on the Status of IEEE* 754.94720-1776 (1996), p. 11 (Cited on page 28).

[97] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, et al. "A study of BFLOAT16 for deep learning training". In: *arXiv preprint arXiv:1905.12322* (2019) (Cited on page 30).

[98] K. Kalliojarvi and J. Astola. "Roundoff errors in block-floating-point systems". In: *IEEE transactions on signal processing* 44.4 (1996), pp. 783–790 (Cited on page 81).

[99] G. Karunaratne, M. Schmuck, M. Le Gallo, G. Cherubini, L. Benini, A. Sebastian, and A. Rahimi. "Robust high-dimensional memory-augmented neural networks". In: *Nature communications* 12.1 (2021), p. 2468 (Cited on page 54).

[100] H. Katebi, N. Asadi, and M. Goudarzi. "FullPack: Full Vector Utilization for Sub-Byte Quantized Vector-Matrix Multiplication on General Purpose CPUs". In: *IEEE Computer Architecture Letters* (2024) (Cited on page 122).

[101] J. Kiefer and J. Wolfowitz. "Stochastic estimation of the maximum of a regression function". In: *The Annals of Mathematical Statistics* (1952), pp. 462–466 (Cited on page 21).

[102] Y. Kim, C. Oh, J. Hwang, W. Kim, S. Oh, Y. Lee, H. Sharma, A. Yazdanbakhsh, and J. Park. "DaCapo: Accelerating Continuous Learning in Autonomous Systems for Video Analytics". In: *arXiv preprint arXiv:2403.14353* (2024) (Cited on pages: 33, 106, 109).

[103] D. P. Kingma. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014) (Cited on page 21).

[104] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge. "Domain-specific architectures: Research problems and promising approaches". In: *ACM Transactions on Embedded Computing Systems* 22.2 (2023), pp. 1–26 (Cited on page 6).

[105] R. Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018) (Cited on page 5).

[106] S. Krishnan, M. Lam, S. Chitlangia, Z. Wan, G. Barth-maron, A. Faust, and V. J. Reddi. "QuaRL: Quantization for Fast and Environmentally Sustainable Reinforcement Learning". In: *Transactions on Machine Learning Research* (2022). ISSN: 2835-8856. URL: `https://openreview.net/forum?id=xwWsiFmUEs` (Cited on page 63).

[107] S. Krishnan, M. Lam, S. Chitlangia, Z. Wan, G. Barth-Maron, A. Faust, and V. J. Reddi. "QuaRL: Quantization for fast and environmentally sustainable reinforcement learning". In: *arXiv preprint arXiv:1910.01055* (2019) (Cited on page 8).

[108] A. Krizhevsky, G. Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009) (Cited on pages: 96, 131).

[109] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012) (Cited on pages: 1, 7, 22).

[110] T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel. "Model-ensemble trust-region policy optimization". In: *arXiv preprint arXiv:1802.10592* (2018) (Cited on page 145).

[111] Y. D. Kwon, R. Li, S. Venieris, J. Chauhan, N. D. Lane, and C. Mascolo. "TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge". In: *Forty-first International Conference on Machine Learning*. 2024 (Cited on pages: 2, 3).

[112] A. F. Laguna, X. Yin, D. Reis, M. Niemier, and X. S. Hu. "Ferroelectric FET based in-memory computing for few-shot learning". In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. 2019, pp. 373–378 (Cited on page 54).

[113] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. "Human-level concept learning through probabilistic program induction". In: *Science* 350.6266 (2015), pp. 1332–1338 (Cited on page 47).

[114] N. Lambert, A. Wilcox, H. Zhang, K. S. Pister, and R. Calandra. "Learning accurate long-term dynamics for model-based reinforcement learning". In: *2021 60th IEEE Conference on decision and control (CDC)*. IEEE. 2021, pp. 2880–2887 (Cited on page 145).

[115] C. R. Lazo, E. Reggiani, C. R. Morales, R. F. Bagué, L. A. V. Vargas, M. A. R. Salinas, M. V. Cortés, O. S. Ünsal, and A. Cristal. "Adaptable register file organization for vector processors". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 786–799 (Cited on page 146).

[116] 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 (1989), pp. 541–551 (Cited on page 22).

[117] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48 (Cited on page 16).

[118] S. Lee, J. Choi, S. Noh, J. Koo, and J. Kung. "DBPS: Dynamic Block Size and Precision Scaling for Efficient DNN Training Supported by RISC-V ISA Extensions". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2023, pp. 1–6 (Cited on pages: 33, 82, 106, 107, 108, 109, 123, 124, 147).

[119] M. Li and X. S. Hu. "A quantization framework for neural network adaption at the edge". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 402–407 (Cited on pages: 54, 55).

[120] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. "Pruning and quantization for deep neural network acceleration: A survey". In: *Neurocomputing* 461 (2021), pp. 370–403 (Cited on page 28).

[121] T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman. "Random synaptic feedback weights support error backpropagation for deep learning". In: *Nature communications* 7.1 (2016), p. 13276 (Cited on pages: 26, 147).

[122] T. P. Lillicrap and A. Santoro. "Backpropagation through time and the brain". In: *Current opinion in neurobiology* 55 (2019), pp. 82–89 (Cited on page 25).

[123] A. Limited. *Arm NN SDK | Efficient ML for Arm CPUs, GPUs, & NPUs*. Accessed: 2024-11-05. 2024. URL: `https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn` (Cited on page 9).

[124] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. "Mcunet: Tiny deep learning on iot devices". In: *Advances in neural information processing systems* 33 (2020), pp. 11711–11722 (Cited on page 23).

[125] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han. "On-device training under 256kb memory". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 22941–22954 (Cited on pages: 9, 21, 109).

[126] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, et al. "Deepseek-v3 technical report". In: *arXiv preprint arXiv:2412.19437* (2024) (Cited on page 147).

[127] D. Liu, H. Kong, X. Luo, W. Liu, and R. Subramaniam. "Bringing AI to edge: From deep learning's perspective". In: *Neurocomputing* (2021) (Cited on page 40).

[128] Y. Liu, Y. Gao, and W. Yin. "An improved analysis of stochastic gradient descent with momentum". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 18261–18271 (Cited on page 21).

[129] Z. Liu, G. Zhou, J. He, T. Marcucci, F.-F. Li, J. Wu, and Y. Li. "Model-based control with sparse neural dynamics". In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 6280–6296 (Cited on page 145).

[130] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, et al. "The gem5 simulator: Version 20.0+". In: *arXiv preprint arXiv:2007.03152* (2020) (Cited on page 116).

[131] A. Ltd. *NEON Instruction Set Architecture*. Accessed: 2024-10-29. ARM Ltd. 2013. URL: `https://developer.arm.com/documentation/den0018/a/NEON-Instruction-Set-Architecture` (Cited on page 9).

[132] C.-H. Lu, Y.-C. Wu, and C.-H. Yang. "A 2.25 TOPS/W fully-integrated deep CNN learning processor with on-chip training". In: *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE. 2019, pp. 65–68 (Cited on pages: 40, 107).

[133] S. Lym, E. Choukse, S. Zangeneh, W. Wen, S. Sanghavi, and M. 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*. 2019, pp. 1–13 (Cited on page 28).

[134] C. Ma, J. Wu, C. Si, and K. C. Tan. "Scaling Supervised Local Learning with Augmented Auxiliary Networks". In: *arXiv preprint arXiv:2402.17318* (2024) (Cited on pages: 26, 147).

[135] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/` (Cited on page 20).

[136] S. Matsuoka. "Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU". In: *2021 Symposium on VLSI Circuits*. 2021, pp. 1–3. DOI: `10.23919/VLSICircuits52068.2021.9492415` (Cited on page 77).

[137] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133 (Cited on page 15).

[138] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, et al. "Mixed precision training". In: *arXiv preprint arXiv:1710.03740* (2017) (Cited on page 21).

[139] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius. "Accelerating sparse deep neural networks". In: *arXiv preprint arXiv:2104.08378* (2021) (Cited on page 147).

[140] N. S. Murthy, N. Laubeuf, D. Bhattacharjee, F. Catthoor, and M. Verhelst. "Adaptive Block-Scaled GeMMs on Vector Processors for DNN Training at the Edge". In: *2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC)*. 2024, pp. 1–6. DOI: `10.1109/VLSI-SoC62099.2024.10767806` (Cited on pages: 12, 107, 142).

[141] N. S. Murthy, P. Vrancx, N. Laubeuf, P. Debacker, F. Catthoor, and M. Verhelst. "Learn to Learn on Chip: Hardware-aware Meta-learning for Quantized Few-shot Learning at the Edge". In: *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. IEEE. 2022, pp. 14–25 (Cited on pages: 11, 40, 140).

[142] D. Nadalini, M. Rusci, L. Benini, and F. Conti. "Reduced precision floating-point optimization for Deep Neural Network On-Device Learning on microcontrollers". In: *Future Generation Computer Systems* 149 (2023), pp. 212–226 (Cited on pages: 9, 109).

[143] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine. "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning". In: *2018 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2018, pp. 7559–7566 (Cited on page 145).

[144] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling. "Data-free quantization through weight equalization and bias correction". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1325–1334 (Cited on page 31).

[145] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, and T. Blankevoort. "A white paper on neural network quantization". In: *arXiv preprint arXiv:2106.08295* (2021) (Cited on pages: 45, 70, 130, 131).

[146] V. Nair and G. E. Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814 (Cited on page 18).

[147] M. G. d. Nascimento, V. A. Prisacariu, R. Fawcett, and M. Langhammer. "HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation". In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2023, pp. 6364–6373 (Cited on page 108).

[148] S. M. Neuman, B. Plancher, B. P. Duisterhof, S. Krishnan, C. Banbury, M. Mazumder, S. Prakash, J. Jabbour, A. Faust, G. C. de Croon, et al. "Tiny robot learning: Challenges and directions for machine learning in resource-constrained robots". In: *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2022, pp. 296–299 (Cited on pages: 60, 107).

[149] R. Neven, M. Verhelst, T. Tuytelaars, and T. Goedemé. "Feed-Forward On-Edge Fine-Tuning Using Static Synthetic Gradient Modules". In: *Computer Vision–ECCV 2020 Workshops: Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*. Springer. 2020, pp. 131–146 (Cited on page 7).

[150] A. Ng. "Unbiggen AI—IEEE Spectrum". In: *URL: https://spectrum. ieee. org/andrew-ng-data-centric-ai (visited on 05/25/2022)* (2022) (Cited on pages: 2, 26).

[151] K. Ni, X. Yin, A. F. Laguna, S. Joshi, S. Dünkel, M. Trentzsch, J. Müller, S. Beyer, M. Niemier, X. S. Hu, et al. "Ferroelectric ternary content-addressable memory for one-shot learning". In: *Nature Electronics* 2.11 (2019), pp. 521–529 (Cited on page 54).

[152] A. Nichol, J. Achiam, and J. Schulman. "On first-order meta-learning algorithms". In: *arXiv preprint arXiv:1803.02999* (2018) (Cited on pages: 7, 40, 41, 42, 43, 47, 48, 149).

[153] S.-H. Noh, J. Koo, S. Lee, J. Park, and J. Kung. "FlexBlock: A Flexible DNN Training Accelerator With Multi-Mode Block Floating Point Support". In: *IEEE Transactions on Computers* 72.9 (2023), pp. 2522–2535. DOI: 10.1109/TC.2023.3253050 (Cited on page 83).

[154] A. Nøkland. "Direct feedback alignment provides learning in deep neural networks". In: *Advances in neural information processing systems* 29 (2016) (Cited on pages: 7, 26, 147).

[155] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson. "The design process for Google's training chips: TPUv2 and TPUv3". In: *IEEE Micro* 41.2 (2021), pp. 56–63 (Cited on page 76).

[156] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright. "Pushing the limits of accelerator efficiency while retaining programmability". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 27–39 (Cited on pages: 6, 138).

[157] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. "Activation functions: Comparison of trends in practice and research for deep learning". In: *arXiv preprint arXiv:1811.03378* (2018) (Cited on page 18).

[158] J. Oh, H. Yoo, C. Kim, and S.-Y. Yun. "Boil: Towards representation change for few-shot learning". In: *arXiv preprint arXiv:2008.08882* (2020) (Cited on page 143).

[159] oneapi-src. *oneDNN*. Accessed: 2024-11-05. 2024. URL: https://github.com/oneapi-src/oneDNN (Cited on pages: 9, 146).

[160] Open Compute Project. *AMD, ARM, Intel, Meta, Microsoft, NVIDIA, and Qualcomm Standardize Next-Generation Narrow Precision Data Formats for AI*. Accessed: 2024-10-29. 2024. URL: https://www.opencompute.org/blog/amd-arm-intel-meta-microsoft-nvidia-and-qualcomm-standardize-next-generation-narrow-precision-data-formats-for-ai (Cited on pages: 9, 30, 34, 138, 144).

[161] A. Oppenheim. "Realization of digital filters using block-floating-point arithmetic". In: *IEEE transactions on audio and electroacoustics* 18.2 (1970), pp. 130–136 (Cited on pages: 7, 30, 32).

[162] A. K. Oudjida. "Binary arithmetic for finite-word-length linear controllers: MEMS applications". PhD thesis. Université de Franche-Comté, 2014 (Cited on page 129).

[163] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. "Training language models to follow instructions with human feedback". In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744 (Cited on page 147).

[164] S. J. Pan and Q. Yang. "A survey on transfer learning". In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359 (Cited on pages: 4, 5, 8).

[165] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter. "Continual lifelong learning with neural networks: A review". In: *Neural networks* 113 (2019), pp. 54–71 (Cited on pages: 5, 143).

[166] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf` (Cited on pages: 20, 47).

[167] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos. "RISC-V 2: a scalable RISC-V vector processor". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5 (Cited on page 77).

[168] M. Pavan, E. Ostrovan, A. Caltabiano, and M. Roveri. "TyBox: An Automatic Design and Code Generation Toolbox for TinyML Incremental On-Device Learning". In: *ACM Transactions on Embedded Computing Systems* 23.3 (2024), pp. 1–27 (Cited on page 3).

[169] A. Pellegrini. "Arm Neoverse N2: Arm's 2 nd generation high performance infrastructure CPUs and system IPs". In: *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE. 2021, pp. 1–27 (Cited on page 77).

[170] A. Pellegrini and C. Abernathy. "Arm neoverse N1 cloud-to-edge infrastructure SoCs". In: *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society. 2019, pp. 1–21 (Cited on page 77).

[171] M. Perotti, M. Cavalcante, R. Andri, L. Cavigelli, and L. Benini. "Ara2: Exploring Single-and Multi-Core Vector Processing with an Efficient RVV 1.0 Compliant Open-Source Processor". In: *IEEE Transactions on Computers* (2024) (Cited on pages: 10, 37, 76, 146).

[172] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, and L. Benini. "A "New Ara" for vector computing: an open source highly efficient RISC-V V 1.0 vector processor design". In: *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2022, pp. 43–51 (Cited on page 76).

[173] C. Pinneri, S. Sawant, S. Blaes, J. Achterhold, J. Stueckler, M. Rolinek, and G. Martius. "Sample-efficient cross-entropy method for real-time planning". In: *Conference on Robot Learning*. PMLR. 2021, pp. 1049–1065 (Cited on page 60).

[174] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe. "Binary neural networks: A survey". In: *Pattern Recognition* 105 (2020), p. 107281 (Cited on page 28).

[175] I. (Qualcomm Innovation Center. *Speculative Decode: AI100 Weight-Only Quantization Using Microscaling MX Formats*. `https://quic.github.io/cloud-ai-sdk-pages/latest/blogs/Speculative_Decode/spec_decode_ai100`. Accessed: 2024-11-19. 2024 (Cited on page 32).

[176] J. von Rad and F. Seuffert. "Investigating the Effect of Network Pruning on Performance and Interpretability". In: *arXiv preprint arXiv:2409.19727* (2024) (Cited on page 28).

[177] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, D. Verkest, and H. Corporaal. "Very wide register: An asymmetric register file organization for low power embedded processors". In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE. 2007, pp. 1–6 (Cited on page 6).

[178] A. Raghu, M. Raghu, S. Bengio, and O. Vinyals. "Rapid learning or feature reuse? towards understanding the effectiveness of maml". In: *arXiv preprint arXiv:1909.09157* (2019) (Cited on page 143).

[179] C. Ramírez, A. Castelló, and E. S. Quintana-Ortí. "A BLIS-like matrix multiplication for machine learning in the RISC-V ISA-based GAP8 processor". In: *The Journal of Supercomputing* 78.16 (2022), pp. 18051–18060 (Cited on page 83).

[180] A. Ranjan, S. Jain, J. R. Stevens, D. Das, B. Kaul, and A. Raghunathan. "X-mann: A crossbar based architecture for memory augmented neural networks". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6 (Cited on page 54).

[181] D. B. Rawat and H. El Alami. "Metaverse: Requirements, architecture, standards, status, challenges, and perspectives". In: *IEEE Internet of Things Magazine* 6.1 (2023), pp. 14–18 (Cited on page 3).

[182] D. Reis, A. F. Laguna, M. Niemier, and X. S. Hu. "A fast and energy efficient computing-in-memory architecture for few-shot learning applications". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 127–132 (Cited on pages: 54, 55).

[183] A. Rico, J. A. Joao, C. Adeniyi-Jones, and E. Van Hensbergen. "ARM HPC Ecosystem and the Reemergence of Vectors". In: *Proceedings of the Computing Frontiers Conference*. 2017, pp. 329–334 (Cited on page 80).

[184] M. Risso, A. Burrello, F. Conti, L. Lamberti, Y. Chen, L. Benini, E. Macii, M. Poncino, and D. J. Pagliari. "Lightweight neural architecture search for temporal convolutional networks at the edge". In: *IEEE Transactions on Computers* 72.3 (2022), pp. 744–758 (Cited on page 5).

[185] B. Rouhani, R. Zhao, V. Elango, R. Shafipour, M. Hall, M. Mesmakhosroshahi, A. More, L. Melnick, M. Golub, G. Varatkar, et al. "Shared Microexponents: A Little Shifting Goes a Long Way". In: *arXiv preprint arXiv:2302.08007* (2023) (Cited on pages: 83, 91).

[186] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf, et al. "Microscaling data formats for deep learning". In: *arXiv preprint arXiv:2310.10537* (2023) (Cited on page 108).

[187] K. Roy, A. Jaiswal, and P. Panda. "Towards spike-based machine intelligence with neuromorphic computing". In: *Nature* 575.7784 (2019), pp. 607–617 (Cited on page 24).

[188] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536 (Cited on pages: 1, 23).

[189] R. M. Russell. "The CRAY-1 computer system". In: *Communications of the ACM* 21.1 (1978), pp. 63–72 (Cited on pages: 37, 146).

[190] C. Sakr, Y. Kim, and N. Shanbhag. "Analytical guarantees on numerical precision of deep neural networks". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 3007–3016 (Cited on page 32).

[191] E. Samson, N. Mellempudi, W. Luk, and G. A. Constantinides. "Exploring FPGA designs for MX and beyond". In: *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2024, pp. 304–310 (Cited on pages: 10, 34, 146).

[192] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520 (Cited on page 131).

[193] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. "Meta-learning with memory-augmented neural networks". In: *International conference on machine learning*. PMLR. 2016, pp. 1842–1850 (Cited on page 40).

[194] N. Satya Murthy, F. Catthoor, and M. Verhelst. "Optimization of block-scaled integer GeMMs for efficient DNN deployment on scalable in-order vector processors". In: *Journal of Systems Architecture* (2024), p. 103236 (Cited on pages: 12, 76, 113, 141).

[195] N. Satya Murthy, F. Catthoor, M. Verhelst, and P. Vrancx. "Quantized Dynamics Models for Hardware-Efficient Control and Planning in Model-Based RL". In: *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Ed. by R. Meo and F. Silvestri. Cham: Springer Nature Switzerland, 2025, pp. 196–209. ISBN: 978-3-031-74643-7 (Cited on pages: 12, 58, 92, 140).

[196] A. W. Services. *AWS DeepLens*. AI-powered video camera for developers, Available at: `https://aws.amazon.com/deeplens/`, Accessed on: October 14, 2024. 2017 (Cited on page 2).

[197] A. W. Services. *AWS Graviton3*. `https://en.wikipedia.org/wiki/AWS_Graviton`. Accessed: 2024-11-27. 2021. URL: `https://en.wikipedia.org/wiki/AWS_Graviton` (Cited on page 77).

[198] C. Shin, J. Lee, B. Na, and S. Yoon. "Personalized Face Authentication Based On Few-Shot Meta-Learning". In: *2021 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2021, pp. 3897–3901 (Cited on page 40).

[199] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) (Cited on page 131).

[200] K. P. Sinaga and M.-S. Yang. "Unsupervised K-means clustering algorithm". In: *IEEE access* 8 (2020), pp. 80716–80727 (Cited on page 24).

[201] J. Snell, K. Swersky, and R. Zemel. "Prototypical networks for few-shot learning". In: *Advances in neural information processing systems* 30 (2017) (Cited on page 40).

[202] I. Soloveychik, I. Lyubomirsky, X. Wang, and S. Bhoja. "Block Format Error Bounds and Optimal Block Size Selection". In: *arXiv preprint arXiv:2210.05470* (2022) (Cited on page 33).

[203] N. Stephens. *ARM architecture developments - ARM v8.6A*. `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-developments-armv8-6-a`. Accessed: 2024-02-01. 2019 (Cited on page 77).

[204] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, et al. "The ARM scalable vector extension". In: *IEEE micro* 37.2 (2017), pp. 26–39 (Cited on pages: 6, 9, 77, 79, 80).

[205] Q. Sun, Y. Liu, T.-S. Chua, and B. Schiele. "Meta-transfer learning for few-shot learning". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 403–412 (Cited on page 143).

[206] R. Sun. "Optimization for deep learning: theory and algorithms". In: *arXiv preprint arXiv:1912.08957* (2019) (Cited on page 21).

[207] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan. "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks". In: *Advances in neural information processing systems* 32 (2019) (Cited on pages: 7, 30, 32, 49, 54).

[208] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan. "Ultra-low precision 4-bit training of deep neural networks". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1796–1807 (Cited on pages: 7, 54).

[209] R. S. Sutton. "Reinforcement learning: An introduction". In: *A Bradford Book* (2018) (Cited on pages: 8, 24, 57, 59).

[210] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329 (Cited on page 5).

[211] H. Tanaka, Y. Takeuchi, K. Sakanushi, M. Imai, and S. Kobayashi. "A block-floating-point processor for rapid application development". In: *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*. Vol. 2. IEEE. 2007, pp. II–65 (Cited on page 32).

[212] M. Tancik, B. Mildenhall, T. Wang, D. Schmidt, P. P. Srinivasan, J. T. Barron, and R. Ng. "Learned initializations for optimizing coordinate-based neural representations". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 2846–2855 (Cited on pages: 3, 144).

[213] H. Tayeb, L. Paillat, and B. Bramas. "Autovesk: Automatic vectorized code generation from unstructured static kernels using graph transformations". In: *ACM Transactions on Architecture and Code Optimization* 21.1 (2023), pp. 1–25 (Cited on page 79).

[214] M. E. Taylor and P. Stone. "Transfer learning for reinforcement learning domains: A survey." In: *Journal of Machine Learning Research* 10.7 (2009) (Cited on page 47).

[215] E. Todorov, T. Erez, and Y. Tassa. "Mujoco: A physics engine for model-based control". In: *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2012, pp. 5026–5033 (Cited on page 66).

[216] B. K. Tripathy, A. Sundareswaran, and S. Ghela. *Unsupervised learning approaches for dimensionality reduction and data visualization*. CRC Press, 2021 (Cited on page 24).

[217] F. G. Van Zee and R. A. Van De Geijn. "BLIS: A framework for rapidly instantiating BLAS functionality". In: *ACM Transactions on Mathematical Software (TOMS)* 41.3 (2015), pp. 1–33 (Cited on pages: 18, 83).

[218] A. Vaswani. "Attention is all you need". In: *Advances in Neural Information Processing Systems* (2017) (Cited on page 23).

[219] M. Verhelst, M. Shi, and L. Mei. "ML processors are going multi-core: A performance dream or a scheduling nightmare?" In: *IEEE Solid-State Circuits Magazine* 14.4 (2022), pp. 18–27 (Cited on page 6).

[220] B.-E. Verhoef, N. Laubeuf, S. Cosemans, P. Debacker, I. Papistas, A. Mallik, and D. Verkest. "FQ-conv: Fully quantized convolution for efficient and accurate inference". In: *arXiv preprint arXiv:1912.09356* (2019) (Cited on page 46).

[221] P. Villalobos, J. Sevilla, L. Heim, T. Besiroglu, M. Hobbhahn, and A. Ho. "Will we run out of data? an analysis of the limits of scaling datasets in machine learning". In: *arXiv preprint arXiv:2211.04325* (2022) (Cited on page 2).

[222] D. Wang, Y. Cheng, M. Yu, X. Guo, and T. Zhang. "A hybrid approach with optimization-based and metric-based meta-learner for few-shot learning". In: *Neurocomputing* 349 (2019), pp. 202–211 (Cited on page 143).

[223] M. Wang, R. Xue, J. Lin, and Z. Wang. "Exploring Quantization in Few-Shot Learning". In: *2020 18th IEEE International New Circuits and Systems Conference (NEWCAS)*. 2020, pp. 279–282. DOI: 10.1109/NEWCAS49341.2020.9159767 (Cited on pages: 54, 55).

[224] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. "Training deep neural networks with 8-bit floating point numbers". In: *Advances in neural information processing systems* 31 (2018) (Cited on page 44).

[225] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba. "Benchmarking model-based reinforcement learning". In: *arXiv preprint arXiv:1907.02057* (2019) (Cited on page 60).

[226] X. Wang, Y. Chen, and W. Zhu. "A survey on curriculum learning". In: *IEEE transactions on pattern analysis and machine intelligence* 44.9 (2021), pp. 4555–4576 (Cited on page 26).

[227] X. Wang, L. L. Zhang, Y. Wang, and M. Yang. "Towards efficient vision transformer inference: A first study of transformers on mobile devices". In: *Proceedings of the 23rd annual international workshop on mobile computing systems and applications*. 2022, pp. 1–7 (Cited on page 23).

[228] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni. "Generalizing from a few examples: A survey on few-shot learning". In: *ACM computing surveys (csur)* 53.3 (2020), pp. 1–34 (Cited on pages: 4, 5, 7, 42).

[229] C. White, W. Neiswanger, and Y. Savani. "Bananas: Bayesian optimization with neural architectures for neural architecture search". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 35. 12. 2021, pp. 10293–10301 (Cited on page 27).

[230] J. Won, J. Si, S. Son, T. J. Ham, and J. W. Lee. "Ulppack: Fast sub-8-bit matrix multiply on commodity simd hardware". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 52–63 (Cited on page 122).

[231] S. Wu, G. Li, F. Chen, and L. Shi. "Training and inference with integers in deep neural networks". In: *arXiv preprint arXiv:1802.04680* (2018) (Cited on pages: 7, 40, 54).

[232] Y. Wu and K. He. "Group normalization". In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 3–19 (Cited on page 27).

[233] L. Wulfert, J. Kühnel, L. Krupp, J. Viga, C. Wiede, P. Gembaczka, and A. Grabmaier. "AIfES: A Next-Generation Edge AI Framework". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024) (Cited on pages: 9, 146).

[234] Z. Xianyi, W. Qian, and Z. Chothia. "OpenBLAS". In: *URL: http://xianyi. github. io/OpenBLAS* 88 (2012) (Cited on page 83).

[235] Y. Xu, Y. Sun, X. Liu, and Y. Zheng. "A digital-twin-assisted fault diagnosis using deep transfer learning". In: *Ieee Access* 7 (2019), pp. 19990–19999 (Cited on pages: 3, 145).

[236] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. De Sa. "SWALP: Stochastic weight averaging in low precision training". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 7015–7024 (Cited on pages: 40, 41, 47).

[237] J. Yang, M. Wen, J. Shen, Y. Cao, M. Tang, R. Yang, J. Fei, and C. Zhang. "BP-im2col: Implicit im2col supporting AI backpropagation on systolic arrays". In: *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE. 2022, pp. 415–418 (Cited on page 20).

[238] T.-J. Yang, Y.-H. Chen, and V. Sze. "Designing energy-efficient convolutional neural networks using energy-aware pruning". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5687–5695 (Cited on page 28).

[239] P. Yin, J. Lyu, S. Zhang, S. Osher, Y. Qi, and J. Xin. "Understanding straight-through estimator in training activation quantized neural nets". In: *arXiv preprint arXiv:1903.05662* (2019) (Cited on page 32).

[240] P. Yu, A. Levisse, M. Gupta, E. Timon, G. Ansaloni, F. Catthoor, and D. Atienza. "A Soft SIMD Based Energy Efficient Computing Microarchitecture". In: *arXiv preprint arXiv:2212.09358* (2022) (Cited on page 38).

[241] P. Yu, F. Ponzina, A. Levisse, D. Biswas, G. Ansaloni, D. Atienza, and F. Catthoor. "DBFS: Dynamic Bitwidth-Frequency Scaling for Efficient Software-defined SIMD". In: *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2024, pp. 204–209 (Cited on page 38).

[242] P. Yu, F. Ponzina, A. Levisse, M. Gupta, D. Biswas, G. Ansaloni, D. Atienza, and F. Catthoor. "An Energy Efficient Soft SIMD Microarchitecture and Its Application on Quantized CNNs". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2024) (Cited on pages: 6, 37, 121, 123, 125, 126, 142).

[243] T. Yu and H. Zhu. "Hyper-parameter optimization: A review of algorithms and applications". In: *arXiv preprint arXiv:2003.05689* (2020) (Cited on page 21).

[244] B. Zhang, R. Rajan, L. Pineda, N. Lambert, A. Biedenkapp, K. Chua, F. Hutter, and R. Calandra. "On the importance of hyperparameter optimization for model-based reinforcement learning". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2021, pp. 4015–4023 (Cited on page 73).

[245] J. Zhang, W. Yu, X. Yang, and F. Deng. "Few-shot learning for ear recognition". In: *Proceedings of the 2019 International Conference on Image, Video and Signal Processing*. 2019, pp. 50–54 (Cited on page 40).

[246] S. Q. Zhang, B. McDanel, and H. Kung. “Fast: DNN training under variable precision block floating point with stochastic rounding”. In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 846–860 (Cited on pages: 33, 106, 107, 108, 109, 123, 124, 147).

[247] T. Zhang, Z. Lin, G. Yang, and C. De Sa. “Qpytorch: A low-precision arithmetic simulation framework”. In: *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE. 2019, pp. 10–13 (Cited on pages: 44, 45, 47, 67, 103, 116, 131).

[248] T. Zhang, C. Lei, Z. Zhang, X.-B. Meng, and C. P. Chen. “AS-NAS: Adaptive scalable neural architecture search with reinforced evolutionary algorithm for deep learning”. In: *IEEE Transactions on Evolutionary Computation* 25.5 (2021), pp. 830–841 (Cited on page 27).

[249] W. Zhang, L. Niu, D. Zhang, G. Wang, F. U. D. Farrukh, and C. Zhang. “Hw-adam: Fpga-based accelerator for adaptive moment estimation”. In: *Electronics* 12.2 (2023), p. 263 (Cited on page 147).

[250] C. Zhou, Q. Li, C. Li, J. Yu, Y. Liu, G. Wang, K. Zhang, C. Ji, Q. Yan, L. He, et al. “A comprehensive survey on pretrained foundation models: A history from bert to chatgpt”. In: *arXiv preprint arXiv:2302.09419* (2023) (Cited on page 23).

[251] S. Zhu, T. Voigt, F. Rahimian, and J. Ko. “On-device Training: A First Overview on Existing Systems”. In: *ACM Transactions on Sensor Networks* 20.6 (2024), pp. 1–39 (Cited on page 108).

Use of Generative AI

I did use the generative AI assistance tools: ChatGPT

***ChatGPT* was used for generating programming code.**

- Used for generating template plotting code with descriptions of formatting.
- Used for organization of data or transformations.
- Used for very quick generic plots to be used as visual examples with dummy data.
- Used for generating Latex code to convert table or equation images to custom modifiable code for use as template for quick processing.

The text/code/ images in this thesis are my own (unless otherwise specified) and generative AI has only been used in accordance with the KU Leuven guidelines and appropriate references have been added. I have reviewed and edited the content as needed and I take full responsibility for the content of the thesis.

Plagiarism prevention, use of Gen AI and ALLEA code of conduct (version October 2024)

List of publications

Articles in international journals

- **Nitish Satya Murthy**, Francky Catthoor, and Marian Verhelst. "Optimization of block-scaled integer GeMMs for efficient DNN deployment on scalable in-order vector processors." Journal of Systems Architecture 154 (2024): 103236.

Articles in international conference proceedings

- **Nitish Satya Murthy**, Peter Vrancx, Nathan Laubeuf, Peter Debacker, Francky Catthoor, and Marian Verhelst. "Learn to Learn on Chip: Hardware-aware Meta-learning for Quantized Few-shot Learning at the Edge." In 2022 IEEE/ACM 7th Symposium on Edge Computing (SEC), pp. 14-25. IEEE, 2022.
- **Nitish Satya Murthy**, Nathan Laubeuf, Debjyoti Bhattacharjee, Francky Catthoor, and Marian Verhelst. "Adaptive block-scaled GeMMs on vector processors for DNN training at the edge". In: 2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC). (Best paper candidate)

Articles in workshops

- **Nitish Satya Murthy**, Francky Catthoor, Marian Verhelst and Peter Vrancx. "Quantized dynamics models for hardware-efficient control and planning in model-based RL". In: Machine Learning and Principles and Practice of Knowledge Discovery in Databases. Ed. by R. Meo and F. Silvestri. Cham:Springer Nature Switzerland, 2025, pp. 196–209.

Poster presentation

- **Nitish Satya Murthy**, Francky Catthoor, and Marian Verhelst. "Optimization of block-scaled GeMM kernels on vector processors", HiPeac summer school – 14-20 July, Fiuggi, Italy, ACACES 2024.



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
ESAT-MICAS | IMEC
Kasteelpark Arenberg 10
B-3001 Leuven
nitish.satyamurthy@kuleuven.be
www.website.kuleuven.be