Diss. ETH No. 26716

The concept of transprecision computing

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

FLORIAN MICHAEL SCHEIDEGGER
MSc ETH EEIT, ETH Zurich
born on April 4th, 1990
citizen of Unterramsern SO, Switzerland

accepted on the recommendation of

Prof. Dr. Luca Benini, examiner
Prof. Dr. Norbert When, co-examiner
Dr. Cristiano Malossi, supervisor

2020

Acknowledgments

First of all, I would like to thank my supervisors, Luca Benini and Cristiano Malossi. I appreciate that I got the exceptional chance to pursue a doctorate within IBM Research Zurich. In that context, I thank Luca for his prompt and constructive feedback and his guidance to higher-level goals. But most importantly, I thank Luca for opening the doors of that career path in its initial stages — without that, I would not be in my current position. I would like to thank Cristiano for his endless support, his detailed feedback, and fruitful discussions. Especially, I thank Cristiano for practical help and guidance required in the daily business, which includes improving presentation skills up to successfully leading me through a complete process of filing a disclosure as the first author. I would like to thank Costas Bekas, our group manager, for pushing through my contract and for providing motivating and constructive feedback to our team. I would also like to thank Norbert Wehn, the co-examiner of this thesis, for his interest in my work and his feedback.

I thank all my collaborators, Goran Flegar, Vedran Novakovic, Giovani Mariani, Andres Tomas, Enrique Quintana-Ortí, Lukas Cavigelli, Michael Schaffner, Roxana Istrate, Dimitrios Nikolopoulos, Youri Popoff, Michael Gautschi, Frank Gürkaynak, Hubert Kaeslin, Aljoscha Smolic, Manuel Eggimann, Christelle Gloor, Atin Sood, Benjamin Elder, Benjamin Herta, Chao Xue, Debashish Saha, Ganesh Venkataraman, Gegi Thomas, Hendrik Strobelt, Horst Samulowitz, Martin Wistuba, Matteo Manica, Mihir Choudhury, Rong Yan, Ruchir Puri, and Tejaswini Pedapati for constructive discussions and inspiration that caused joint contributions.

I thank all members of the OPRECOMP consortium for impacting our research. Especially, I would like to thank Enrique Quintana-Ortí, Andrew Emerson, Fabian Schuiki, Stefan Mach, Guiseppe, Davide Rossi, Giuseppe Tagliavini, Andrea Borghesi, JunKyu Lee, Umar Minhas, Igor Neri, Eric Flamand, and Christian Weis for their valuable project input, critical feedback and long discussions that help to shape this work. I would like to thank additional team members at IBM, Nico Gorbach, Panagiotis Chatzidoukas, and Andrea Bartezzaghi that were involved in productive technical discussions. A special thank is dedicated to Roxana with whom I shared the office, for technical discussions, for helping me out with many practical problems, and for the joint work that we authored.

I would like to thank Luca's research group and IBM's department for providing the facility and required administrative help. A special thank is dedicated to Christoph Hagleitner for managing and providing IBM's computing infrastructure to researchers. I highly appreciate the proficient help of Lilli-Marie Pavka for copy-editing our latest publication. I thank Dionysios Diamantopoulos for interesting technical out-of-the-box discussions and shared OPRECOMP project efforts.

I would like to thank Folashade Ajala, Austin Brauser, and Hari Mistry for volunteering to proofread fragments of this work.

Finally, I would like to thank my family and my friends for supporting me during this time. I highly appreciate the rooted to the soil inspiration of Sarah Battige that calmed me down in stressful times towards the end of writing this work. I am in particular grateful to my parents Marianne and Thomas Scheidegger who provide unconditional support in any life situations. Especially, I would like to thank them for encouraging my interests and for providing me access to the university in the first place that enabled my career path.

Abstract

For many years, computing systems rely on guaranteed numerical precision of each step in complex computations. Moore's law sustains exponential improvements in the semiconductor industry over several decades for building computing infrastructure, from tiny Internet-of-Things nodes, over personal smartphones, laptops or workstations, up to large high performance computing (HPC) computing server centers. With the paradigm of the "power wall", achievable improvements start to saturate. To that end, the concept of transprecision computing emerged, where existing over-conservative "precis" computing assumptions are relaxed and replaced with more flexible and efficient policies to gain performance.

Unfortunately, it is non-straight forward to adopt and integrate general transprecision concepts into the variety of today's computing infrastructure. The main challenge consists of leveraging domain-specific knowledge and provide full solutions covering from physical foundations over circuit-level up through the full software stack to the application level.

This work focuses on how transprecision concepts improve general computing. We identify and elaborate the standard number representations, especially the one defined in the IEEE 754 floating-point standard, as the enabler of low precision computing. We developed lightweight libraries that allow integrating transprecision concepts into algorithms. Finally, we focus on building automatized workflows for specific problems, where the solution space is enlarged by multiple orders of magnitude due to the various configurations of low precision. We demonstrate how heuristic optimization strategies applied on top

of transprecision computing find near to optimal configurations of approximated kernels in a short time.

Zusammenfassung

Seit vielen Jahren beruhen komplexen Berechnungen von Computer Systemen auf garantierter numerischer Präzision in jedem Schritt. Das Mooresche Gesetz (engl. Moore's law) erhält exponentielles Wachstum in der Halbleiterindustrie über mehrere Jahrzehnte aufrecht. Dadurch wird ein kontinuierlicher Fortschritt von Computer Infrastruktur — vom Internet der Dinge, Smartphones, Laptops, Arbeitsplatzrechnern, bis hin zu Hochleistungsrechnern in Rechenzentren — erreicht. Das Paradigma der Grenzen der Leistungsaufnahme (engl. power wall) limitiert erreichbare Verbesserungen. Das Konzept von Transprecision Computing lockert existierende und zu konservative Annahmen bezüglich der Rechengenauigkeit. Stattdessen werden Annahmen durch flexiblere und effizientere Richtlinien ersetzt um die Rechenleistung zu verbessern.

Leider ist es nicht einfach Transprecision Konzepte direkt in die Vielfalt der heutigen Computer Systeme zu integrieren. Die grösste Herausforderung besteht darin, Gebiet spezifisches Wissen wirksam einzusetzen, um eine komplette Lösung zu erreichen welche alle Aspekte — von physikalischen Grundlagen, Schaltungsdetails, Software, bis hin zu Anwendungsgegebenheiten — berücksichtigt.

Diese Arbeit fokussiert wie durch Transprecision Konzepte allgemeine Berechnungen verbessert werden können. Wir identifizieren und etablieren, dass die Standard Repräsentation von Zahlen, insbesondre jene des IEEE 754 floating-point Standards, das Rechnen mit

reduzierter Genauigkeit ermöglichen. Wir entwickeln schlanke Softwarebibliotheken welche die Integration von Transprecision Konzepte in Algorithmen ermöglichen. Schlussendlich bilden wir automatisierte Arbeitsabläufe für spezifische Problemstellungen welche aufgrund der vielen Konfigurationen der einstellbaren Genauigkeit in einem, um mehrere Grössenordnungen erweiterten, Lösungsraum liegen. Wir zeigen, wie heuristische Optimierungsstrategien — angewandt auf Genauigkeits-Konfigurationen von Transprecision Berechnungen — nahezu optimale Konfigurationen der approximierten Kernen in kurzer Zeit liefern.

Contents

Chapter 1

Introduction

Driven by Moore's law, technology scaling has consistently supported energy-aware computing over the last decades [1]. However, physical brick walls such as heat removal and signal propagation delays limit the system operating frequency. Figure 1.1 shows the number of transistors for central processing units (CPUs), graphical processing units (GPUs), and field programmable gate arrays (FPGAs) dependent on the year of introduction[1]. Impressively, the scaling follows a steady exponential growth that has been maintained over the past four decades. In contrast, Figure 1.2 shows the operating frequency that starts to saturate[2]. Until the turn of the millennium, the frequency of ancient computing devices improved exponentially. More recent systems exceed an operation frequency of $1\,GHz$ but nowadays common systems typically operated at a frequency around $2\,GHz$ and $4\,GHz$.

The quest for performance and energy efficiency remains. Various ways of parallelism (i.e., multicore, single instruction multiple data (SIMD)/Vector, single instruction multiple threads (SIMT)), specialized application-specific integrated circuit (ASIC) accelerators, and FPGA implementations are developed to meet the requirements.

[1] Available at `https://en.wikipedia.org/wiki/Transistor_count` (Accessed November 2019)

[2] Available at `https://en.wikipedia.org/wiki/Microprocessor_chronology` (Accessed November 2019)

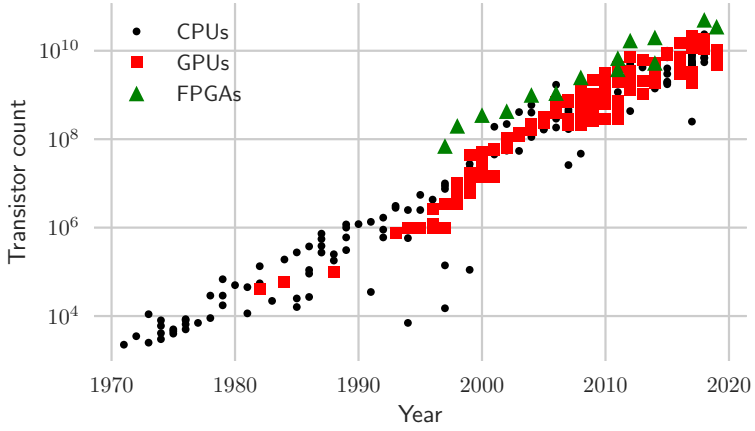


Figure 1.1: Moore's law. The number of transistors present in integrated chips grows exponentially over time. The same observation holds for regular processors, GPUs and FPGAs.

Orthogonal to such approaches, *transprecision computing* considers the number representation as an additional degree of freedom to gain efficiency from more compact floating-point formats. Transprecision computing improves systems due to reducing the number representation and the related arithmetic. Simultaneously, transprecision computing systems ensure the quality of the final results.

In this work, we elaborate on how transprecision concepts affect *general* computing applications. We provide promising results on three applications based on our developed transprecision concepts including search algorithms and reduced precision libraries. We demonstrate that PageRank [2], an iterative algorithm, can replace a majority of computations into compact floating-point representation, while still converging to the same quality of the final result as the baseline. bidirectional LSTM (BLSTM) [3] profits from compact floating-point formats with negligible impacts on the final accuracy. The Gauss-Legendre quadrature (GLQ) implements a typical routine used in

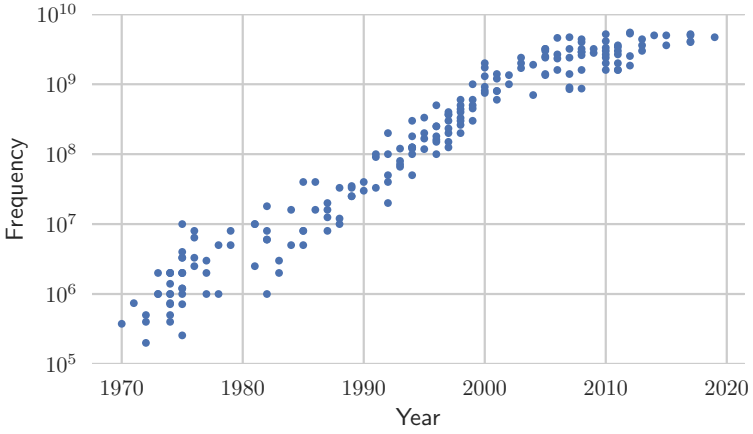


Figure 1.2: The system clock scaled well before the millennium. Nowadays, the system frequency saturates around $2\,GHz$ and $4\,GHz$ due to physical limitations and economical limitations.

scientific computing. Even without degenerating results, reduced precision can be used during intermediate representations.

Materializing a vision by developing components of a new computing paradigm is an exciting and challenging task. The true value of transprecision computing is achievable when multiple parties collaborate by independently developing products that follow the new paradigm. We expect that the expertise and efforts from various contributors multiply the success of the *transprecision computing*. However, the initial steps bring several engineering and research challenges before new approaches become mainstream. We formulate long and short term goals of *transprecision computing*.

In the long-term, *transprecision computing* exploits approximation in both hardware and software to boost energy efficiency [4]. Moore's law drives the semiconductor industry for general-purpose processors independently from traditional software development. In contrast, *transprecision computing* relies on a thigh coupling between hardware and software to further improve the energy efficiency of the final

application or systems. The rapidly developing research area known as approximate computing [5–7] pursues the same goals. Malossi et al. [4] envision that *transprecision computing* progresses the state-of-the-art along several axes:

1. It controls approximation in space and time (when and where) at a fine grain through multiple hardware and software feedback control loops.
2. It does not imply reduced precision at the application level, even though it is also possible to exploit application-level softening of precision requirements for extra benefits.
3. It takes inspiration from nature by defining computing architectures that operate with a smooth and wide range of precision vs. cost trade-off curve.

The key short term goals that we address in this thesis are:

1. Formulate and define the concepts of transprecision computing;
2. Demonstrate the success of reduced precision for various computing domains;
3. Improve aspects of generality, scalability, and simplicity to apply the concepts;
4. Elaborate detailed considerations of transprecision computing in the domain of deep learning.

Progressing the paradigm of transprecision computing consists of achieving high-level goals. In Section 1.4 we outline the structure of the thesis by explaining how the content contributes to reaching the meta goals.

In the next section, we present a dedicated introduction to deep learning. We decided to highlight that topic due to the following reasons. First, domain knowledge is assumed to better understand the contributions in Chapter 6 and Chapter 8. Second, even by considering regular arithmetic only, we achieved an outstanding contribution Chapter 8 by developing a constrains neural network search.

Table 1.1: Key components of a supervised machine learning workflow.

$\mathcal{T}$: Task	$\mathcal{D}$: Dataset	$\mathcal{A}$: Algorithm	$\mathcal{M}$: Model
Classification	Images	ML algorithm	Building blocks
Segmentation	Videos	DL algorithm	DNN topology
Object detection	Audio	Transfer learning	Parameters
Forecasting	Text	Few-shot learning	Preprocessing
[...]	Tabular data	[...]	[...]
	Time series		
	[...]		

Third, to better understand the scaling and generality properties of *transprecision computing* it is as important to ensure that concepts are applied and evaluated with the latest state-of-the-art as the developing of the concept itself. The fast pace and ongoing progress in deep learning favor the development of modular and reusable *transprecision computing* components.

1.1 Trends in deep learning

Deep learning is considered one of the most promising solutions for solving machine learning problems. At the time of writing, a search with the keyword *"deep learning"* returns over four and a half million publications on Google Scholar, out of which a substantial part of 10% are published within the current year. The extensive literature, the numerous deep learning (DL) competitions, and available open-source resources further demonstrate the wide interest in the topic. This introduction covers the full deep learning era to justify in which environment contributions of this thesis are embedded. We focus on supervised learning problems that are based on a labeled dataset that provides ground truth.

Table 1.1 states the key elements that compose a machine learning workflow and lists common choices for each element. $\mathcal{T}$ denotes a deep learning task on an input dataset $\mathcal{D}$, by running an algorithm $\mathcal{A}$ and resulting in a trained model $\mathcal{M}$. The dataset is a collection of paired

input $\mathbf{x}$ and output y samples $\mathcal{D} := (\mathbf{x}, y)$ and the model $\mathcal{M}$ is a mapping from the input to the output space $\mathcal{M} : \mathbf{x} \rightarrow y$ optimized to solve the given task $\mathcal{T}$. Supervised ML algorithms learn relations in the annotated dataset between the raw input data and the assigned labels. Non neural network ML approaches include linear classifiers [8], K-nearest neighbors (KNN), [9], support vector machines (SVM) [10], Random Forests [11], and similar methods. DL approaches are the subset of ML approaches that consist of neural networks.

Over the last years, two key enablers have driven the success of deep learning. First, the availability of large scale datasets with known ground truth [12–20] enables supervised learning for complex tasks such as face recognition [12, 13], action recognition [14, 15] or video classification [19–21]. Second, the availability of increased computational performance in today's computing systems typically achieved with GPUs enables to train large scale models. Figure 1.3 states launch year and performance of Nvidia Tesla products, Nvidia's product line that is tailored to serve the high-performance computing market domain for general-purpose GPU computing. The Tesla product line is missing some of the traditional graphics peripherals in favor of using the full chip area for computing. Such GPU device configurations are widely used in large scale data centers, cloud computing environments, supercomputers, or research cluster settings. Figure 1.3 highlights three products, the K80, P100, and V100 GPUs that were used to perform GPU related workloads presented in this thesis. The historical analysis shows, that GPU hardware followed a constant improvement over the past decade. The S870 GPU, the best available option that was available back in 2007 when the Tesla product line started, achieves a single-precision performance of 1.38 TFLOPS. Ten years later, in 2017, the V100 GPU was launched that is able to deliver 14.03 TFLOPS amounting to a 10× improvement in performance over a decade.

At the same time, deep learning research has evolved rapidly. Motivated by improved results and success, researchers extended the focus in specialized subdomains of deep learning. Deep learning methods rely on the availability of a full ecosystem including hardware, a deep learning framework, and the core work of the data scientist achieving solutions for specific tasks. The common interest of different stockholders, such as industries working on applications powered by

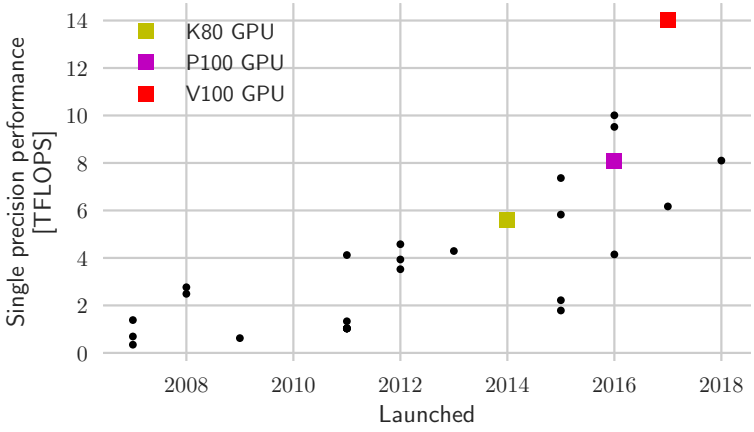


Figure 1.3: GPU performance of the NVIDIA Tesla series.

deep learning, hardware designers, manufacturers, and traditional research fuels the motivation and drives the current progress in the field into various directions. We look at current and historical results achieved on the CIFAR10 dataset [22] to provide an overview of the trend that is happening in the field of deep learning. CIFAR10 is an instance of an image classification problem with 10 classes. The dataset provides 50,000 training samples and marks 10,000 samples for testing. We measure the progress based on 63 publications. Figure 1.4 states the achieved Top1 accuracy on CIFAR10 and the year of publication. The per-year-mean Top1 accuracy increased from 80.5% to 98.5% from 2011 till 2019. In other words, the quality increased by about two percentage points per year. By definition, the accuracy cannot exceed 100% which builds a natural upper bound and causes accuracy to saturate. Due to the short interval on which state-of-the-art remains, development and research become challenging since any performed comparisons are likely to get outdated in a short time. In the next sections, we outline research subdomains of deep learning and we highlight the scope to which we have extensively studied transprecision computing concepts.

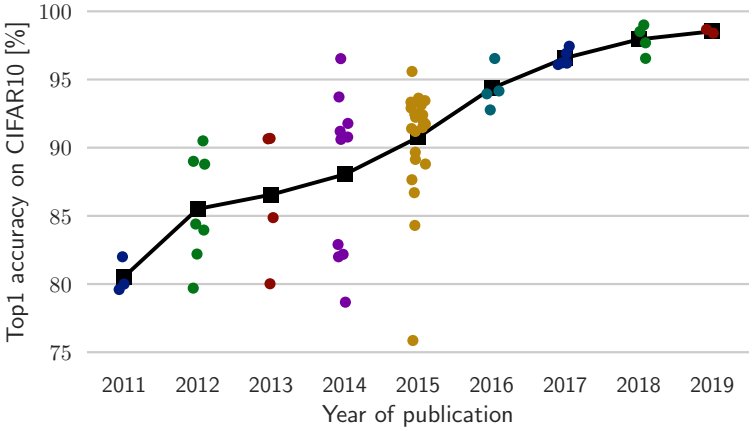


Figure 1.4: Top1 accuracy achieved of 63 reference methods on CIFAR10. Developers and researchers constantly improved results over one decade.

1.2 Research directions in deep learning

The main development steps through which a data scientist works to implement deep learning solutions are:

1. Data acquisition and collection,
2. Data annotation and cleaning (Labour intense step),
3. Model definition (Performance critical decision),
4. Algorithm definition (Performance critical decision),
5. Model training (Resource and time-consuming step),
6. Model evaluation (including hyper parameter optimization),
7. Repeat and refine steps.

Step 1 to Step 2 are related to obtaining a large dataset. Step 1 deals with how data is captured including information about sensor resolution and exposure times. Collection deals with how multiple samples are obtained, i.e., how many different sources are used and how data was merged. Step 2 deals with annotating and cleaning the data. In many cases, researchers can access a full dataset that has been collected, cleaned, annotated and is provided to the public, such as the CIFAR10 dataset [22]. Depending on the use-case specific settings, annotating and cleaning the data is a labor-intense process. In this thesis, we assume that the annotated dataset is available and we focus on the active side of deep learning. In the following, we summarize key research subdomains that are all related to the common deep learning workflow.

The core machine learning part consists of designing a model in Step 3, selecting an algorithm in Step 4 and training the model on the given data in Step 5. Final results are evaluated and optimized in Step 6. Typically, depending on the results, parts of this workflow are fine-tuned and repeated to improve results. The following sections summarise research areas that affect the pipeline at different stages.

Model design

Model design solves the problem of designing or selecting a convolutional neural network architecture that defines a parametric function that maps inputs to outputs. The field evolves and manual designed network architectures improved results over the past years. For example, first success with convolutional neural networks was achieved with VGG [23] that follows a simple pattern of a sequence of convolutional layers. Later, ResNets [24] were introduced, which use residual connections inside the topology that enables to train deeper neural networks that are more accurate. More recent architectures, such as Inception [25], dual-path networks (DPN) [26], or DenseNets [27] follow all more complex design patterns with high fan-out and fan-in branching occurring in the network graph. Different networks introduced novel design patterns for different purposes, for example, MobileNet [28] was designed to reduce memory usage and execution time. MobileNet achieves that by factorizing traditional 3×3 convolutions into a depthwise and 1×1 pointwise convolution that improves

computational cost of about 8 to 9 times and empirically demonstrates to achieve good accuracies. Over time, smaller changes in the network structure were proposed and heavily used, such as different activation functions, global-max pooling operations, and regularization operations that are coupled with the training algorithm such as dropout and batch normalization layers.

Neural architecture search (NAS)

Since defining a good model is non-trivial early results of automated neural architecture search (NAS) were motivated to potentially discovers better models [29–35]. However, traditional approaches require a vast amount of computing resources or cause excessive execution times due to the full training of candidate networks [36]. Follow-up work optimizes the required search time by shortening or avoiding expensive candidate network training times [37–39]. However, due to the success of manual designed topologies that are developed independently, architecture searches face the common challenge of defining the search space. Method evaluations are criticized since even for experienced researchers it becomes difficult to understand if performance improvements are achieved due to the search algorithm within a given space or due to extensions of the space [40]. The latest developments are going in the direction to adapt the search algorithm and its setup to optimize for a specific task, for example to speed-up the inference time of a neural network running on a smartphone [41, 42].

Learning algorithms

The predominant learning algorithms for neural networks are (variants of) stochastic gradient descent. Implementations thereof rely on computing the derivative of the loss with respect to the trainable parameters such that they can be updated to minimize the error. Research deals with how to initialize weights and how to add regularization to the learning [43]. Optimizers such as Adadelta [44], Adagrad [45], Adam, or AdaMax [46] improve learning behaviour by adapting learning rates and using different variants of weight updates. Different researchers looked into 2nd order methods and applied them

in the context of image classification, such as the Tonga [47] algorithm. However, traditional 2nd order methods, such as Newton's method or the Conjugate gradient method require to compute the inverse Hessian and the additional complexity causes larger memory requirements and intractable computational burdens [48]. Overall, surprisingly many research papers use the vanilla stochastic gradient descent (SGD) implementation with a learning rate schedule to achieve the most accurate results even though it is recommended to use and adaptive method for fast convergence [49].

Hyperparameter optimization

Hyperparameter optimization (HPO) deals with the problem of optimizing non-trainable parametric values that are used throughout the deep learning workflow. Early work in the field considered selecting the correct network topology as HPO problem, however, we think that it is worth to separately study the NAS problem since the high interest and dedicated solutions targeting that problem explicitly as explained before. However, even if the network architecture is fixed, there are many settings a data scientist has to decide without a clear answer to what works best can be given upfront. For example, the training algorithm as previously explained can be considered as a categorical choice for an HPO algorithm. More typical settings that could be exposed to an HPO algorithm included the learning rate, optimizer specific settings, learning rate scheduling strategies, settings for preprocessing, and data augmentation policies. Simple HPO algorithms operate as Grid or Random search [50]. Bayesian optimization [51,52] tries to first test configurations in unknown domains that are likely to provide good results. Hyperband optimization [53] improves time costs to solution quality by balancing the amount of tried configurations versus the time spend to test one configuration by reducing the number of alive configurations during training and removing bad candidates early in the process. HPO might improve model performance but is limited by the choice on what parametric values it is applied and it consumes a large number of resources for a single optimization task.

Data augmentation

Data augmentation aims to extend the training dataset by modifying existing samples to improve the model generalization performance. Standard techniques include on-the-fly applied image transformations such as random flips, random crops, color-, brightness-, sharpness-adaptions, blurring, adding noise, stretching, and rotating images. Different methods account for dealing with unbalanced datasets. Majority class under-sampling, Minority class over-sampling, or using generative adversarial networks (GANs) to synthesize augmented data are among the common solutions to tackle class unbalances [54]. Some datasets are provided with a fixed resolution (for example CIFAR10 [22] is provided with 32×32 pixels) and others are not. In the latter case, using resizing and cropping combinations to obtain the resolution that is feed into the neural network provides additional degrees of freedom on what scale the deep learning model is operated.

Domain adaption

Domain adaption deals with questions around multiple datasets and multiple deep learning workflows. The term transfer learning [55] refers to research problems related to take insights from experiments or results obtained on a *source* dataset $\mathcal{D}_S$ over to a new *target* dataset $\mathcal{D}_T$. Reusing a model trained on the source dataset on the target dataset by initializing the weights or by freezing layers and only fine-tuning a few dense layers at the tail of the network potentially provides three advantages against a from scratch trained model: first, the warm-start provides already more accurate results, second, during training the models follows a learning curve that outperforms the baseline, and third, the model potentially saturates at a higher accuracy. Especially, for smaller datasets, freezing layers that have learned to extract useful features avoids over-fitting and helps to improve the generalization error on the target dataset. In a similar context, *few-shot classification* [56], the problem of learning to generalize to unseen classes during training from a few annotated samples, has gained popularity to account for the fact large annotated datasets from domain-specific problems are not available, or time-consuming and expensive to gather.

Optimizing for speed

State-of-the-art neural networks (NNs) for image classification typically have 10-200 million parameters and require 10-25 billion arithmetic operations to perform inference for a single image [57]. Such deep networks achieve high classification accuracies, but also require long training times [58]. While the race to improve accuracy on challenges such as the imagenet - large scale visual recognition challenge (ILSVRC) [59] drives the community to develop ever more complex models, this trend is likely to continue with the increasing availability of video-based datasets. For these NNs to remain economically viable, it is important to keep the computational effort in mind to reduce the costs involved when building practical systems for large-scale inference, such as energy and infrastructure expenditures [60]. Research directions aiming to improve time-to-solution or to outperform a baseline in at least one key cost factor, such as energy to solution, are manifold. For example, model parallelism and distributed training deal with the question of how to extend algorithms on multi compute resources. Additional, there is a heavy industrial interest in how to build specialized hardware for deep learning, from ASIC design, FPGA based solutions up to providing and extending machine learning solutions of existing general-purpose computing systems.

ML ecosystems

ML research is performed at the top of the iceberg of a full software and hardware stack. The problem of consideration implies a relying underlying environment that implements and runs standard functionality. Common deep learning frameworks include TensorFlow [61], PyTorch [62], Chainer [63], MXNet [64] among many more. All of them implement common standard functionality, such as defining neural networks, setting up optimizers and training models with a variety of algorithms and they support GPU acceleration for all core functionality. Since deep learning research, especially work in the subdomain of the NAS problem, includes to run many training runs, the availability of multi-node resources equipped with a queueing, scheduling, and managing system is required. Many Cloud providers

offer to rent customized hardware with pre-installed software that is tailored to serve the need of the deep learning market.

1.3 Contributions

The key contributions of this thesis are summarized below:

- **Providing an abstraction of transprecision computing.**

  We formalize transprecision computing and related problems, including characterization, configuration search, and design space exploration. We focus on the key notation of quality versus performance trade-offs.

- **Designing, benchmarking, and implementing a reduced precision library.**

  We contributed the core quantization routines that emulate the low-level behavior of reduced floating-point formats. We verified the behavior of floatx by exhaustive tests that compare floatx half implementations against third party implementations.

- **Integrating transprecision into PyTorch.**

  We discussed the integration of reduced precision into PyTorch [62], a commonly used deep learning framework. We provide analytical and empirical arguments that compare intrinsic and extrinsic emulation approaches. We implemented several utility functions that help to traverse, insert, and modify computational graphs to produce all results presented in this work.

- **Producing error-resilience results for reference models.**

  We claim the generality of transprecision computing. We support that argument by running numerical experiments on 30

well-established behavior. In accordance with reported error-resilience of deep learning models, our experiments confirm the general applicability of transprecision computing.

- **A constraint NAS algorithm for the IoT.**

  We contributed narrow-space NAS synthesized models that meet constraints. With a simple device-specific calibration approach we are able to generate models meeting inference time requirement on low-cost internet of things (IoT) devices. Our search enables to produce customized models with limited budget during the synthesis and a resource limited IoT device.

- **A large-scale reduced precision study for IoT.**

  We used our NAS approach to fully train over 3,000 baseline models. Applying transprecision computing to the full set of models enables to get global insight. We observed that in terms of weight memory footprint versus accuracy, transprecision outperforms regular models with a wide margin.

- **Discussing and developing practical aspects around transprecision computing.**

  We contribute research and engineering efforts into developing transprecision computing and related practical aspects. We provide heuristic approaches to study the configuration problem that enhances the search speed significantly. Additionally, we ensured that our implementation of the quantization operation integrated into PyTorch [62] executes on the GPU. Joining those efforts allows performing more experiments in a shorter time. Short evaluation cycles are a key factor to apply the discussed concepts in practice on novel problems.

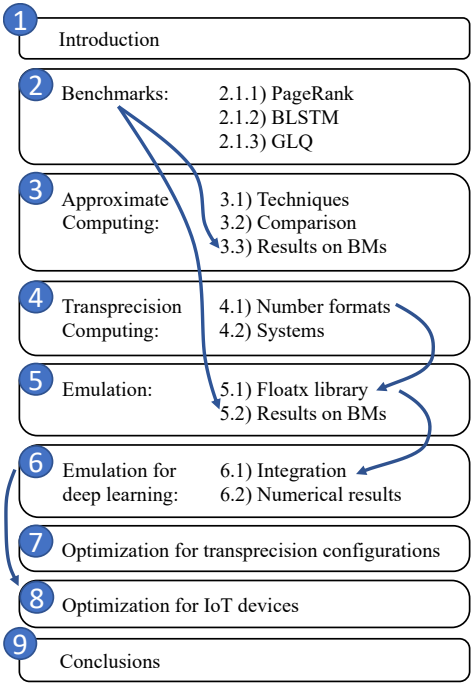


Figure 1.5: Thesis structure.

1.4 Thesis structure

Figure 1.5 shows an overview of the structure of the thesis. Chapter 2 starts the thesis by summarizing some of the existing benchmarks and their use. We identify representative candidates from three different computing domains to cover the variability of applications that occur in general-purpose computing. Section 2.1.1, Section 2.1.2, and Section 2.1.3, present PageRank, BLSTM, and GLQ as examples of Big-Data, deep learning, and Scientific computing. Selecting applications from three different domains contributes towards the generality claim of transprecision computing as stated in Item 3 and realizes the goal defined in Item 2. We use those applications throughout the thesis to demonstrate high-level considerations of *transprecision computing*.

Chapter 3 summarizes established approximate computing approaches. We identified the most promising technique in Section 3.2 and we apply it to applications in Section 3.3. We identified common challenges among such approaches, such as strict data dependency and the notion of application quality. We observe that the definition of acceptable quality gets a major role to reveal the potential for performance improvements. Additionally, we observe that inherently existing hyper-parameters already provide a competitive way to trade-off quality versus performance in many cases.

Chapter 4 defines the concepts of transprecision computing and focuses on the goal stated in Item 1. We discuss number representations in Section 4.1 and define *transprecision computing* on a conceptual level in Section 4.2.1. We focus on the notation of quality and performance depending on a transprecision configuration space. The abstraction leaves room for future systems to be exploited with the same mentality, while we already provide answers to related engineering problems. We define the configuration space design, the characterization, and the configuration optimization as reusable meta problems.

Chapter 5 designs and implements a C++ library that allows emulating reduced floating-point formats. We explain design choices of floatx in Section 5.1.2. Section 5.1.2 provides low-level implementations details and Section 5.1.8 discusses the scaling and performance of the library. Section 5.2 uses floatx to provide in-depth numerical studies of the three applications.

Chapter 6 deals with the integration of floatx into PyTorch [62], a commonly used deep learning framework. We discuss emulation aspects and conditions for which a fast and efficient numerical emulation is achievable in Section 6. We demonstrate scaling and generality of numerical evaluations by defining 30 well-established reference image classification models in Section 6.2.1 and we report numerical results in Section 6.2.2. The advances reported are specific for the domain of deep learning, covering the goal stated in Item 4. The concept of transprecision demonstrates to be model agnostic and follows similar behaviors for different reference models, henceforth, that supports the claimed generality of transprecision computing requested in Item 3.

Chapter 7 and Chapter 8 focuses on automatization and optimization of workflows. Section 7.2 introduces specific search heuristics to

efficiently find good enough *transprecision* configurations in reasonable time. Our proposed algorithms demonstrate in Section 7.4 how the search times are significantly reduced. Those insights build the fundament of simplification to quickly apply *transprecision computing* to new problems as requested in Item 3.

In Chapter 8 we develop a constrained neural network architecture search tailored for the Internet-of-Things. Even in the regular domain of using full 32-bit precision our approach is novel and outperforms alternatives. Additionally, the search involves the training of over 3,000 models that enable a large-scale exploration of reduced precision computing. To the best of our knowledge, we are the first that demonstrate the excellent behavior of models operating with reduced precision. Even when opportunity costs are considered in the comparison, that involves synthesizing models operating with full precision, the reduced precision models provide a better overall trade-off.

Chapter 9 concludes the thesis. We summarize the main findings in Section 9.1 and we provide an outlook on future work in Section 9.2

The extra Chapter A in the appendix discusses a system design of a video classification system. We evaluate three state-of-the-art neural-network-based approaches for large-scale video classification, where the computational efficiency of the inference step is of particular importance due to the ever-increasing amount of data throughput for video streams. Our evaluation focuses on finding good efficiency vs. accuracy tradeoffs by evaluating different network configurations and parametrizations.

1.5 List of publications

Key material covered in this thesis has been published:

[65] F. Scheidegger, L. Benini, C. Bekas, and C. Malossi, "Constrained deep neural network architecture search for iot devices accounting for hardware calibration," in *Advances in Neural Information Processing Systems*, 2019, to appear

[66] G. Flegar, F. Scheidegger, V. Novakovic, G. Mariani, A. Tomas, C. Malossi, and E. Quintana-Ortí, "Float x: A c++library for

customized floating-point arithmetic," *ACM Trans. Math. Softw.*, 2019, to appear

[67] F. Scheidegger, L. Cavigelli, M. Schaffner, A. Malossi, C. Bekas, and L. Benini, "Impact of temporal subsampling on accuracy and performance in practical video classification," in *Signal Processing Conference (EUSIPCO), 2017 25th European.* IEEE, 2017, pp. 996–1000

We have worked and contributed in the domain of deep learning with IBM colleges on the following publications:

[68] F. Scheidegger, R. Istrate, G. Mariani, L. Benini, C. Bekas, and C. Malossi, "Efficient image dataset classification difficulty estimation for predicting deep-learning accuracy," *arXiv preprint arXiv:1803.09588*, 2018

[54] G. Mariani, F. Scheidegger, R. Istrate, C. Bekas, and C. Malossi, "Bagan: Data augmentation with balancing gan," *arXiv preprint arXiv:1803.09655*, 2018

[69] R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, and A. C. I. Malossi, "Tapas: Train-less accuracy predictor for architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 3927–3934

[70] A. Sood, B. Elder, B. Herta, C. Xue, C. Bekas, A. C. I. Malossi, D. Saha, F. Scheidegger, G. Venkataraman, G. Thomas *et al.*, "Neunets: An automated synthesis engine for neural network design," *arXiv preprint arXiv:1901.06261*, 2019

Those works are not explicitly covered in this thesis. However, through long discussions I gained my expertise in deep learning that shapes and supports key contributions of this thesis. Without, that prior learning, the key IoT contribution in Chapter 8 would not have been possible.

Prior to joining IBM, while being employed at ETH, I have worked and contributed to the following publications:

[71] Y. Popoff, F. Scheidegger, M. Schaffner, M. Gautschi, F. K. Gürkaynak, and L. Benini, "High-efficiency logarithmic number

unit design based on an improved cotransformation scheme," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1387–1392

[72] M. Schaffner, F. Scheidegger, L. Cavigelli, H. Kaeslin, L. Benini, and A. Smolic, "Towards edge-aware spatio-temporal filtering in real-time," *IEEE Transactions on Image Processing*, vol. 27, no. 1, pp. 265–280, 2018

[73] M. Eggimann, C. Gloor, F. Scheidegger, L. Cavigelli, M. Schaffner, A. Smolic, and L. Benini, "Hydra: An accelerator for real-time edge-aware permeability filtering in 65nm cmos," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–5

Chapter 2

Benchmarking today's systems

Benchmarks get developed reliably measure applications, systems, software, and hardware components. Well-established benchmarks such as Whetstone [74] and Dhrystone [75] measure and compare early computing systems. Those benchmarks are synthetic, which means that they execute a random instruction-mix that follows the distribution of instructions caused by real applications. Whetstone is designed for benchmarking floating-point performance, Dhrystone benchmarks integer arithmetic.

The standard performance evaluation corporation (SPEC)[1] maintains and releases various benchmark suites. They cover domains including cloud service, CPU performance, workstation performance, E-mail server operation, and Java client/server applications. In contrast to synthetic benchmarks, the SPEC benchmarks [76] perform *useful* payload computations. The setup validates results generated during measurement. Recent versions of the SPEC, such as the SPEC CPU 2017[2], includes 43 tasks to cover a wide variety of applications and use-cases. Different benchmarks, such as the linear algebra package (LINPACK) benchmark [77], assess floating-point performance

[1] Available at `https://www.spec.org/` (October 2019)
[2] Available at `https://www.spec.org/cpu2017` (October 2019)

by solving a system of linear equations. The LINPACK benchmark attracts attention since its use of ranking supercomputers, published as the Top500[3]. Its simplicity and the easy adaptation to new systems explains the success of the LINPACK benchmark. However, the LINPACK benchmark has several limitations, such as the regular memory access patterns do not fully stress the caching and memory subsystems. Additionally, relying on benchmarking one application involves the risk that systems are engineered to over-tune the benchmark metrics. Instead, benchmarks should assess the behavior of real applications for which systems are built.

Similar to the Top500, the Green500[4] ranks today's supercomputers based on energy efficiency [78]. The high performance conjugate gradient (HPCG)[5] benchmark [79] was introduced to overcome some of the limitations of LINPACK. HPCG performs Conjugate Gradient iterations as core operations for measuring performance. The occurring sparsity patterns ensure that non-trivial memory patterns stress the memory subsystem. The authors claim that HPCG performs more meaningful computations that are better correlated with workload distributions appearing in real-world applications.

2.1 Benchmarks for transprecision computing

Approximation computing (see Chapter 3) and transprecision computing (see Chapter 4) rely on two characteristics; performance and quality of the results. To benchmark and quantify such systems, it is essential to measure both. Traditional benchmarks aim to measure the performance of different computing systems. We required benchmarks, that additionally report the quality of results. Even though the newer benchmarks all include result validation procedures, their purpose is to cross-check the full workflow. In contrast, we need a way to measure the output quality of systems that are built with the intent of changing the working precision. Benchmarking

[3] Available at `https://www.top500.org/` (October 2019)
[4] Available at `http://www.green500.org/` (December 2019)
[5] Available at `http://www.hpcg-benchmark.org/` (October 2019)

transprecision systems is challenging since it covers interacting aspects from software, hardware, and compiler stack at multiple granularity levels. Well-written benchmarks target to disentangle influences stemming from different components such as hardware, software, and compilers. Traditional benchmarks are distributed as plain source code, whereas the required tooling, such as the compilation, is left to the target system. That way, those benchmarks use fixed code and minimal compilation requirements to benchmark the hardware systems. In contrast, transprecision benchmarks focus to assess applications and kernels directly. They should characterize the effect of reduced precision and identify reduced precision operation without or with only minor quality degeneration. Benchmarking should strive for a top-down approach including end-to-end behaviors that help to understand the collaborative operation of multiple components.

To demonstrate the concept of transprecision computing, high-level application scalability studies assess the generality and error resilience of applications. Insights and conclusions should be collected before triggering the engineering and designing of customized hardware implementations. From that perspective, the first key question is how much precision reductions do applications allow without harming quality. Addressing the precision-based quality characterization is novel when compared against traditional performance benchmarks. Methodologically, the simplest approach is building an end-to-end system and to measure quality and performance thereon. However, designing a full transprecision system including hardware, programming languages and compilers is out of the scope of this thesis. Instead, we split the evaluation process into two modular and decoupled conceptual steps. First, we use emulation to judge the quality of solutions independent of the performance. Second, we use proxy metrics to estimate the performance. That approach allows postponing the full development of transprecision hardware. Still, it provides a strong methodology to systematically assess transprecision concepts. The modular approach allows to focus on specific components and to reuse invested engineering time. This key insight motivates us to write a reduced precision floating-point library as explained in Chapter 5.

Since understanding numerical effects includes application knowledge, we discuss transprecision on three kernels. We selected representative candidates from three different domains that compute

results with domain-specific meanings. Section 2.1.1, Section 2.1.2, and Section 2.1.3, present PageRank, BLSTM, and GLQ as examples of Big-Data, deep learning, and Scientific computing.

2.1.1 PageRank

PageRank [2] is one of the early building blocks used in web search engines and enabled the success of Google [80]. The idea behind PageRank is that the citation graph of web pages intuitively defines the importance of web pages. Serving web queries had become an integral part for professional and private users, including different domains such as research, industry and private use. The IEEE international conference on data minin (ICDM) selected PageRank in December 2006 among the top-ten data mining algorithms [81]. The fact that Google annually handled more than two trillion queries (the measurement was performed in 2016)[6] impressively demonstrates the relevance of large-scale ranking systems.

PageRank iteratively computes the node score given the topology of a directed graph as the sparse adjacency matrix of size $n \times n$ with z nonzeros. PageRank is known to be numerical stable [82]. Sparsity-aware implementations ($n < z < n^2$) reach a time complexity of $O(Iz)$ and the memory complexity is $O(z)$, where I denotes the input data dependent number of iterations till convergence. PageRank is memory-bound since each iteration accesses z matrix entries while performing constant work $O(1)$ per entry. PageRank is improved with a RAM aware implementation [83] or by changing the algorithm to a graph aggregation technique [84]. Algebraic methods enhance the convergence rate [85].

Algorithm 1 implements PageRank. Line 2 initializes the iteration vector of length n with an uniform distribution where we used that notation $\mathbf{e} = [1, 1, 1, ..., 1]$. The main loop of PageRank iterates until convergence. The iteration vector $\mathbf{p}$ is updated in line 5 where a damping factor $d \in (0, 1]$ is used to stabilize the convergence. Convergence is reached when the iteration vector remains. The final result is normalized distribution that ranks nodes according importance. The

[6] Available at `https://searchengineland.com/google-now-handles-2-999-trillion-searches-per-year-250247` (November 2019)

number of iterations of PageRank depend on the input data $\mathbf{A}$, the damping factor d, and stopping threshold ϵ.

Algorithm 1 PageRank

procedure PAGERANK($\mathbf{A}, d, \epsilon$) ▷ Graph is stored in $n \times n$ sparse and normalized adjacency matrix $\mathbf{A}$
  $\mathbf{p}_0 \leftarrow \mathbf{e}/n$
  $k \leftarrow 1$
  repeat
    $\mathbf{p}_k \leftarrow (1-d)\mathbf{e} + d\mathbf{A}^T\mathbf{p}_{k-1}$
    $k \leftarrow k+1$
  until $\|\mathbf{p}_k - \mathbf{p}_{k-1}\|_1 < \epsilon$
  return $\mathbf{p}_k$ ▷ resulting ranking
end procedure

2.1.2 BLSTM

BLSTM stands for Bidirectional Long Short-Term Memory and refers to topology structures that are used for classifying sequential data. In our context, we refer with BLSTM to the use-case that implements a specific BLSTM topology for solving the optical character recognition (OCR) of old German text (Fraktur) [86]. The use-case with the original reference C-implementation is provided by Rybalkin and Wehn [3] who implemented the first field-programmable gate array (FGPA) acceleration of BLSTM. OCR converts handwritten or printed images of text automatically into machine-encoded text. OCR has applications in various field, first, it acts as data entry of business documents, such as recognizing passports, invoices, bank statements, and receipts. Second, OCR is used in automatic number plate detection systems [87] to identify cars, for example, to provide restricted access to parking lots. Third, OCR helps to ingest information in computer systems in a searchable form by recognizing the text of scanned books [88].

Deep learning models raised to the most successful methods of solving such tasks [89]. Long-short-term memorys (LSTMs) [90] enable the handling of sequential data, occurring in video classification (see Section A), language learning, audio processing, and optical character recognition. The Bidirectional variant for OCR [91] is improved

by a forward backward algorithm, known as connectionist temporal classification (CTC) [92]. CTC is capable of recognizing text lines without requiring any pre-segmentation.

The BLSTM kernel solves the optical character recognition OCR problem on the old German text [3]. Algorithm 2 states the recurrent equations of an LSTM cell [90] that resembles a finite state machine (FSM). The $\odot$ operator denotes element-wise multiplication and σ, h, and g denote element-wise applied activation functions. The logistic sigmoid function is used for σ and the hyperbolic tangent for h and$g$. Input, forget and output gates control the data flow by either passing or attenuating corresponding values. The gates themselves depend on the current input $\mathbf{x}^t$, the last computed output $\mathbf{y}^{t-1}$, and the last state $\mathbf{c}^{t-1}$ and the current state in case of the output gate. In line 5 the main recurrence updates $\mathbf{c}^t$ based on the previous state $\mathbf{c}^{t-1}$, the currently processed input $\mathbf{z}^t$ and the input gate activations i^t and forget gate activations f^t. The final output $\mathbf{y}^t$ is based on the processed state $\mathbf{c}^t$ and the output gate activations $\mathbf{o}^t$, see line 7.

Algorithm 2 LSTM cell update

▷ Trainable parameters of the LSTM cell:
- Weight matrices: $\mathbf{W}_z$, $\mathbf{W}_i$, $\mathbf{W}_f$, $\mathbf{W}_o$, $\mathbf{R}_z$, $\mathbf{R}_i$, $\mathbf{R}_f$, and $\mathbf{R}_o$
- Bias vectors: $\mathbf{b}_z$, $\mathbf{b}_i$, $\mathbf{b}_f$, and $\mathbf{b}_o$
- Peephole selection vectors: $\mathbf{p}_i$, $\mathbf{p}_f$, and $\mathbf{p}_o$

▷ Inputs of the LSTM cell:
- Input vector at time step t is $\mathbf{x}^t$
- Internal state of previous time step $\mathbf{c}^{t-1}$
- Output of previous time step $\mathbf{y}^{t-1}$

1: **procedure** LSTMCellUpdate($\mathbf{x}^t, \mathbf{c}^{t-1}, \mathbf{y}^{t-1}$)
2: $\quad\mathbf{z}^t \leftarrow g(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z)$ ▷ block input
3: $\quad\mathbf{i}^t \leftarrow \sigma(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i)$ ▷ input gate
4: $\quad\mathbf{f}^t \leftarrow \sigma(\mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f)$ ▷ forget gate
5: $\quad\mathbf{c}^t \leftarrow \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t$ ▷ internal state update
6: $\quad\mathbf{o}^t \leftarrow \sigma(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^{t-1} + \mathbf{b}_o)$ ▷ output gate
7: $\quad\mathbf{y}^t \leftarrow h(\mathbf{c}^t) \odot \mathbf{o}^t$ ▷ block output
8: $\quad$**return** $\mathbf{c}_t$, $\mathbf{y}_t$ ▷ return current state and output
9: **end procedure**

Algorithm 3 implements BLSTM. It accepts as input a variable-length sequence of fixed-sized dimensions of vectors, such as the pixel values of the column-wise split input image. First, the LSTM cells are instantiated by loading their internal parameters as used in Algorithm 2. BLSTM performs a forward and backward pass through the two LSTM cells over the scanned input image. Finally, the CTC merges independent partial sequenced in the final output. CTC first concatenates the two sequences and applies a dense layer to compute the logits, see Line 14. Second, the softmax is applied to produce the normalized probabilities and the index of the maximum entry is identified. Third, each index corresponds to a character of the alphabet $\mathcal{A}$, and Line 16 maps indices to characters.

Algorithm 3 BLSTM

1: **procedure** BLSTM($\mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3, ..., \mathbf{x}^n$)
2: $cell_{FW} \leftarrow$ InitFWLSTMCell() ▷ Load model
3: $cell_{BW} \leftarrow$ InitBWLSTMCell()
4: $\mathbf{W}_{CTC},\ \mathcal{A} \leftarrow$ Init()
▷ Forward pass over sequence
5: $\mathbf{c},\ \mathbf{y}^0_{FW} \leftarrow 0$
6: **for** $t \leftarrow 1, n$ **do**
7: $\mathbf{y}^t_{FW}, \mathbf{c} \leftarrow cell_{FW}$.LSTMCellUpdate($\mathbf{x}^t, \mathbf{c}, \mathbf{y}^{t-1}_{FW}$)
8: **end for**
▷ Backward pass over sequence
9: $\mathbf{c},\ \mathbf{y}^{n+1}_{BW} \leftarrow 0$
10: **for** $t \leftarrow 1, n$ **do**
11: $\mathbf{y}^{n+1-t}_{BW}, \mathbf{c} \leftarrow cell_{BW}$.LSTMCellUpdate($\mathbf{x}^{n+1-t}, \mathbf{c}, \mathbf{y}^{n+2-t}_{BW}$)
12: **end for**
▷ CTC pass over sequence
13: **for** $t \leftarrow 1, n$ **do**
14: $\mathbf{l}^t \leftarrow \mathbf{W}_{CTC}[\mathbf{y}^t_{FW}; \mathbf{y}^t_{BW}]$ ▷ compute logits
15: $j^t \leftarrow \arg\max_j softmax(\mathbf{l}^t)$ ▷ maximize index
16: $\alpha^t \leftarrow \mathcal{A}(j^t)$ ▷ map index to character
17: **end for**
18: **return** $\alpha^1, \alpha^2, \alpha^3, ..., \alpha^n$ ▷ return character sequence
19: **end procedure**

In our evaluation setting, the pre-trained BLSTM model infers 3,401 images of text fragments. Samples have a fixed height of 25 pixels and a variable width between 64 and 732 pixels with a mean value of 520.5 pixels and a standard deviation of 100.7 pixels. The BLSTM achieves a Levenshtein distance [93] based accuracy of 98.2337%.

2.1.3 GLQ

Finite element methods (FEMs) provide numerical solutions to the problem of partial differential equations occurring in various applications [94]. FEM applications include magnetic potential modeling [95], heat transfer [96], fluid flow [97], or structural analysis [98]. Those methods build an integral part of industrial engineering. Simulations allow providing insights into problems where the analytic solution does not exist. Finite element methods rely on numerical integration. The GLQ solves the 1D numerical integration problem [99–101]. GLQ extends to the 2D case and can be limited to triangular integration regions [102], which occur as the typical use-case in FEM kernels. In this work, we focus on the 1D GLQ kernel to isolate a relevant part of FEM simulations that cover various applications.

GLQ solves a numerical integration over an arbitrary integrable function. It approximates the integral by a weighted sum of function-evaluations at a set of discrete points. The basic GLQ rule is formulated over the domain $[-1, 1]$ and given as follows:

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i), \tag{2.1}$$

where the weights w_i and positions x_i are precomputed. x_i is the i-th root of the associated Legendre Polynomial $P_n(x)$ and w_i is defined through:

$$w_i = \frac{2}{(1 - x_i^2)(P_n'(x_i)^2)}. \tag{2.2}$$

With a change of variable, the rule extends to work on arbitrary integral domains $[a, b]$. Alternatively, the arbitrary domain integral can be segmented into multiple smaller intervals. Independent GLQ

computations are applied to shifted versions of the original function. We based our reference code on a freely available implementation[7].

We use the six Genz functions [103] for evaluation of the quality of the GLQ implementation. First, Genz functions are analytically integrable such that the closed-form of the result exists. Second, they provide several levels of difficulty such as smooth, less-smooth, continuous, and discontinuous functions. Those use-cases cover typical issues with numerical integration routines. Third, Genz functions are well-established and they are designed for evaluating quadrature procedures [103]. The Genz functions are defined as follows:

$$
\begin{aligned}
f_1(\mathbf{x}) &= \cos\left(2\pi\mathbf{u}_i + \sum_{i=1}^{n} \mathbf{a}_i\mathbf{x}_i\right) \\
f_2(\mathbf{x}) &= \prod_{i=1}^{n} \left(\mathbf{a}_i^{-2} + (\mathbf{x}_i - \mathbf{u}_i)^2)\right)^{-1} \\
f_3(\mathbf{x}) &= \left(1 + \sum_{i=1}^{n} \mathbf{a}_i\mathbf{x}_i\right)^{-(n+1)} \\
f_4(\mathbf{x}) &= \exp\left(-\sum_{i=1}^{n} \mathbf{a}_i^2(\mathbf{x}_i - \mathbf{u}_i)^2\right) \\
f_5(\mathbf{x}) &= \exp\left(-\sum_{i=1}^{n} \mathbf{a}_i\,|\mathbf{x}_i - \mathbf{u}_i)|\right) \\
f_6(\mathbf{x}) &= \begin{cases} 0 & \text{if } \mathbf{x}_1 > \mathbf{u}_1 \text{ of } \mathbf{x}_2 > \mathbf{u}_2 \\ \exp\left(\sum_{i=1}^{n} \mathbf{a}_i\mathbf{x}_i\right) & \text{else} \end{cases}
\end{aligned}
\tag{2.3}
$$

where the parameters $\mathbf{a}$, $\mathbf{u}$, and input $\mathbf{x}$ are vectors of dimension n.

2.2 Summary and conclusion

The most important findings of this chapter are the following:

[7] Available at `http://www.holoborodko.com/pavel/numerical-methods/numerical-integration/l` (November 2019)

- We discussed existing benchmarks, such as Whetstone [74] and Dhrystone [75]. They are traditional synthetic benchmarks that measure floating-point and integer arithmetic performance. The SPEC benchmarks [76] consist of suites of kernels that were designed to provide workloads that occur in practical applications. The linpack benchmark [77] solves a linear system of equations. Due its simplicity and the freedom of fine-tuning source code, it is widely used to rank high-performance systems. However, due to the continuous memory access patterns the benchmark got criticized. The HPCG benchmark [79] solves a sparse Conjugate Gradient iterations that causes non-trivial memory access patterns. HPCG has been proposed as replacement for linpack. HPCG stresses more the memory subsystem and henceforth, better measures the performance that matter for real applications.

- Traditional benchmarks measure the performance of a computing system. To study transprecision concepts we focus on applications an their algorithms. Additional to performance, application specific quality is required to be assessed. We suggest to asses three kernels in detail: PageRank, BLSTM, and GLQ.

- **PageRank**. PageRank is memory-bound and iteratively computes the node score given the topology of a directed graph as sparse adjacency matrix. It become popular for ranking web-pages by using hyperlink information.

- **BLSTM**. BLSTM solves the OCR problem for old German text (Fraktur).

- **GLQ.** The GLQ numerically computes the integral over a function. GLQ is inherently approximative by nature, even when running with full precision, the quality of the integration routine depends on the number of support points.

Chapter 3

Approximate computing

Approximate computing refers to techniques that trade-off achieved quality against the effort spent to obtain results [5–7, 104]. Potentially, inaccurate results are achieved and approximation techniques are suggested to be applied to applications where approximate results are acceptable. However, due to the growing performance demand of applications, approximate computing offers a promising option that is orthogonal to the traditional advances in technology. Approximate computing techniques are effective at various levels.

First, approximate circuits study hardware design trade-offs. Simple circuits predict the output of traditionally slow operating paths to reduce the clock frequency below worst-case propagation paths [105]. Pruned adders are designed to shorten long carry-chains with minimal statistical effects on results [106, 107].

Second, approximate storage considers trade-offs obtained for memory systems. For example, reducing the number of programming pulses improves the write performance of solid-state memories [108]. Similarly, reducing refresh rates of dynamic random access memorys (DRAMs) lowers the standby power consumption of those memories if a small number of failures is allowed [109].

Third, software level approximation changes algorithmic behavior to achieve quality versus performance trade-offs. For example, loop perforation reduces the computation workload by skipping some iterations [110]. Memoization stores the results of function calls for later

reuse with identical function inputs. Approximate computing results rely on the fact that one pre-computed result is used for approximating the result of similar input values. Henceforth, the time that would be required to evaluate the function is omitted [111]. Using multiple inexact program versions provides a choice to trade-off quality versus performance [112].

Fourth, approximate systems apply approximation concepts to various components at the same time. For example, a smart camera system used simultaneously approximation techniques at sensing, memory system and computational level to significantly reduced consumed energy [113].

In this chapter, we provide the insight that the simpler approximation techniques are, the easier they integrate into applications. Additionally, the same ideas apply to platforms such as GPU, FPGA, or ASIC implementations. Using the correct datatype or loop perforation requires little modification to the source. In principle, the generality allows those techniques to be applied to any application. However, in some cases, the degeneration of the approximate result might be so large that it produces unacceptable results.

We observed that many applications inherently depend on input parameters that provide a tuning knob between quality and execution speed. For example, the stopping criterion used in iterative refinements, the number of epochs in a deep learning training, or the number of intervals and the order of the polynomial used in GLQ provide parameters that control the achieved quality. Control loop related parameters are already present in the baseline implementation. Henceforth, tuning the quality trade-off point cannot be attributed to a specific approximation technique. Still, changing the control loop related operation point of the baseline represents an effective and attractive way to obtain Pareto optimal solutions. Those first-order effects provide alternative solutions for comparing approximation results.

We conclude that the choice of numerical representation constructs an effective and general approach. We formalize the concept of transprecision computing in Chapter 4 where we provide an overview of numerical formats in Section 4.1. We identify reduced precision as root-cause for transprecision systems as explained in Section 4.2.2.

3.1 Approximate computing techniques

3.1.1 The use of datatypes

Choosing suitable datatypes for computations improves program execution time. For example, Yeh et al. [114] evaluated the effect of a hierarchical floating-point unit for physical simulations. They concluded that significant improvements are achievable through three reasons: First, quantizing data in reduced representations increases the amount of fast-executing trivial operations. Second, coarser quantized data improves the operation of memoization techniques and in the case of narrow formats, look-up tables outperform the alternative implementations. Third, smaller and faster floating-point units (FPUs) can serve reduced precision computations.

Tian et al. [115] approximate the memory access by using reduced precision. They demonstrate energy savings with a negligible impact on the quality of clustering algorithms.

Floating-point and integer data cover the predominate binary representations of numeric values. In most cases, the former results in using the IEEE 754 64-bit double or 32-bit float standard [116]. Compared with integer arithmetic, floating-point representations provide a high dynamic range. Knowing the upper bounds of integer data allow values to fit into smaller datatypes. However, determining the effect of reducing floating-point width is non-trivial since the precision and the dynamic range are simultaneously affected. Most systems (partially) support integers of widths 8, 16, 32, or 64 bits by densely packing values together to reduce memory footprints and transmission costs. Some systems provide vectorized instructions to support the most common arithmetic operations. In contrast, floating-point arithmetic is either not supported for efficiency reasons (for example, on some microprocessors) or it follows the strict 32-bit or 64-bit IEEE 754 standard on most CPU based systems. Recent GPU hardware supports the IEEE 754 16-bit half datatype and demonstrates how improved floating-point computations scale to the mass market.

However, due to additional programming effort and due to the exponential growth of the configuration space, developers normally avoid writing mixed-precision variants of their applications. Instead, they rely on the 32-bit or 64-bit IEEE 754 floating-point standard.

The conservative policy omits potential gains that are invoked with the half format. Choosing number formats in applications is a general approach that applies to almost all kernels operating with numerical values.

3.1.2 Loop perforation

Loop perforation reduces the total workload of a kernel by executing a subset of iterations of a loop. Perforating source code provides different results when comparing against the original baseline. Thus, loop perforation reduces computation time at the cost of changing the quality of the result. For applications where computations are error-resilient to some degree, loop perforation produces a family of optimal trade-offs. Following the classification by Sidiroglou-Douskos et al. [110], we distinguish between *critical* and *tunable* loops. *Critical* loops must be left unmodified since any change causes unacceptable behavior of the code, such as crashing the application. *Tunable* loops instead offer the potential for perforation. Typically, basic compute patterns that work well with loop perforation are summations, min, max, argmin and argmax patterns [117, 118]. The literature provides examples of those with the probabilistic guarantee that the approximated result is likely to be close to the original result [110, 117, 118]. Loop perforation demonstrated success with the following five global computational patterns [110]:

- search space enumeration,
- search metric,
- Monte-Carlo simulation,
- Iterative improvement,
- Data structure update.

Notation and definition

Without loss of generality, loops are assumed to be in canonical form. They start counting at zero, incrementing by one, and repeat the

loop n times. Henceforth, the canonical loop in the baseline has the following form: *for(i=0; i<n; i++){...}*.

Loop perforation modifies the canonical loop as follows: *for(i=0; i<n; i+=s){...}*, where s denotes the stride factor that controls the skipping. We define the perforation rate r as the expected percentage of skipped loop iterations $r = 1 - 1/s$. For loops where the body is independent of the iteration count, the estimated time is proportional to $1/s$. In cases of independent nested loops, an overall multiplicative gain is achieved for the innermost body fragment. Since loop perforation changes the output, it is important to assess the quality.

Control loops in baseline algorithms

We observe that many applications consist of loops that we classify as *control loops*. The number of iterations is either controlled by a hyper-parameter or dependent on the data. PageRank consists of a residual controlled main loop that acts as a data-dependent control loop. GLQ consists of two nested hyper-parameter controlled loops. One loop iterates the subintervals, the second depends on the polynomial order used to approximate the integral. *Control loops* inherently affect quality and performance. Henceforth, they offer a direct trade-off. We argue, that instead of applying loop perforation to control loops, they should be independently studied. First, since control loops exist in baseline algorithms, understanding their effect is part of the configuration study of kernels. Second, we think that it is unfair to attribute gains obtained by tuning control loops to loop perforation. Third, in some cases, it does not make sense to apply loop perforation. For example, data-dependent termination conditioned loops, are unable to profit from skipping iterations. In other cases, it is better to limit the total number of iterations, rather than skipping iterations. For example, instead of using a GLQ run of order n with a stride factor of $s = 2$, it is more natural to use the GLQ of order $n/2$ without skipping iterations.

Often, users do not fine-tune control loop related parameters. They rely on default, proposed by others, or conservative values. However, control loops provide a very effective way of generating competitive solutions. First, no additional modifications are required and, second, performance gains are achieved on any hardware. We

think that it is essential to study control loops jointly with other approximation techniques due to their advantages. Tuning control loops builds a trivial alternative solution against which approximation techniques must outperform to demonstrate success. In other words, approximations obtained with control loops act as opportunity costs.

3.1.3 Task skipping and memoization

Task skipping and memoization approximation techniques skip memory access, subsample the input or avoid computation of several tasks. These methods rely on the fact that some applications operate on large data chunks that obey a similar statistic. Henceforth, subsampling the input might provide an acceptable quality at better performance. Similarly, stencil-based approximation techniques avoid reading neighboring cells in an image-processing algorithm and reuse a close value instead. Those techniques succeed since in images over 75% of neighboring pixels differ by less than 10% (Figure 5 in [119]). Task skipping and memoization techniques have successfully demonstrated an average speedup of 2.7×. Quality metrics on a GPU for error-resilient, data-parallel applications are guaranteed to have degenerations less of than 10% [119]. The authors have identified six relevant data-parallel patterns that are suitable candidates for approximation:

- Map: a function applied independently to elements of an array.
- Scatter/Gather: like map, but generates non-homogenous memory access patterns.
- Reduction: combines elements of an array to a single output.
- Scan: associative function applied to an input array generating an output array where the result at the n-th position depends on the intermediate state of position $(n-1)$.
- Stencil: a function that is applied to a neighborhood of elements.
- Partition: like stencils, but partitions are present where outputs can be computed independently for partitioned inputs.

In the same study [119], four different approximation techniques were proposed for the six patterns:

- Memoization with lookup tables (Map, Scatter/Gather): instead of evaluating functions, an offline computed lookup table stores results for an application relevant range.
- Subsampling input (Reduction): applies the reduction to a subset of the input data like loop perforation.
- Scan: applies partial scans and omits the computation of several partial scan operations.
- Memory access skipping (Stencil and Partition): based on the assumption that neighboring elements have the same value and the multi-memory reads can be approximated by fewer memory accesses.

3.1.4 Using multiple inexact program versions

For some applications, alternative implementations exist. Each implementation provides a different trade-off in terms of quality and performance. The Pareto optimal choices are suitable candidates for serving the needs of users. For example, based on user requirements and system status the run time selects the most suited kernel. In this way, it is possible to switch to low-quality when a mobile device is operating in low-power mode to extend the battery lifetime.

The identification of the best operating point becomes a run-time optimization problem. For example, Samadi et al. [112] implemented a framework that operates in two phases. First, in a calibration phase data is collected to map quality of service to kernel compositions. Second, the calibration data is used to select—at runtime—the best operation point statistically satisfying target quality requests. The authors propose to recalibrate during regular operation to account for statistical changes occurring in the data. The additional required time is small and allows the technique to be more robust. Similarly, Baek and Chilimbi [120] approximate functions and loops. They suggest a framework that calibrates and operates alternative solutions that provide statistical quality guarantees.

The offline characterizations and run-time optimization concepts are orthogonal to the approximate techniques applied for a single kernel. They become interesting for larger applications where alternatives of composed modules are evaluated. However, they rely on established approximation techniques used within the kernels. Henceforth, they do not provide new concepts of basic approximation techniques.

3.1.5 Stochastic computing

Stochastic Computing represents numbers as probabilities on random bitstreams [121]. The unconventional approach allows building low-complexity hardware that results in low-cost implementations and low power requirements. Since computations in stochastic computing rely on random bitstreams, solutions are inherently error-resilient. However, the are not deterministic. Results are either fully correct or distorted by different quality levels. Caused by the randomness, all possible outputs are obtained with certain probabilities. To understand stochastic computing systems, they are required to be characterized by the probability density function over the output quality. Stochastic computing systems are based on three blocks:

- Binary-to-stochastic: stochastic number generator (SNG).
- Computation unit: very low complexity single gate implementations.
- Stochastic-to-binary: back conversion to traditional binary number with counters.

Stochastic number generators produce randomized bitstreams that represent processed values in the form of the probability that the value one is occurring within the bitstream. The design of stochastic number generators includes studying full applications since correlations between number generators might affect or degenerate results. Surprisingly, although not using true randomness, pseudorandom number generators have been demonstrated to work well. They are traditionally based on linear feedback shift register (LFSR) [122] implementations. Reproducibility, efficient implementations, and stochastic properties to ensure integrity required in crypto applications triggered

extensive research around LFSR generators [123] and variants of alternative hardware friendly random number implementations [124].

Arithmetic operations with stochastic bitstreams operate at low costs. For example, multiplication simplifies to an AND-gate that merges two bitstreams that result in a bitstream that represents the product. Stochastic computing requires normalized values in the interval $[0, 1]$ to map to probabilities. Since adding two numbers results in a larger range, special add-operations are defined that scale the result by a factor of two. This procedure ensures the closeness of the addition operation. Stochastic computing supports elementary operations such as addition, subtraction, and multiplication. Additionally, it has been applied to division and square roots. General-purpose stochastic computing progressed by demonstrating that any given function can be evaluated. This is achieved by approximating a Bernstein polynomial [125] where the coefficients and evaluations of the polynomial are performed with stochastic computing.

Resulting bitstreams require conversion blocks to translate the result back into the traditional binary representation of the number. Simple circuitry achieves that functionality by implementing a counter. Counting is a way to get a statistical estimate of the expected value that conveys the represented value.

The inherently stochastic behavior of stochastic computing provides error tolerance but generates results following a characteristic probability density function over all possible results. Consequently, the output is correct only with a given probability or it might be approximate. In traditional binary representations (fixed-point or floating-point representations, see Section 4.1) the positions of bit-flip errors affect the error of the result, e.g., changing the most significant bit has a larger impact than changing the least significant bit. In contrast, bit flips altering stochastic computing bitstreams cause homogeneous error effects and are not position-dependent. By construction, stochastic computing systems tolerate a few bit-flip errors with only negligible effects on the output. The core arithmetic that implements the computing circuitry has constant complexity, independent of the bitstream length. Since the stream length affects probabilities, more accurate results are achieved when using longer sequences. However, this effect tends to grow exponentially for increased precision requirements leading to large execution times.

Table 3.1: Overview of approximation techniques. We classify methods according being deterministic, if supported on current software and hardware, and ranked overall including expected applicability and performance gains.

Method	DT [1]	LP [2]			TSM [3]						IPV [4]	SC [5]
Deter.	✓	✓			✓						✓	-
SW/HW	**	***			**						**	-
Rating	***	**			*						*	*
Kernel		Control	Tunable	Pattern	Map	Scatter/Gather	Reduction	Scan	Stencil	Partition		
PageRank	✓	1	2	itr.	-	-	✓	-	-	✓	-	-
BLSTM	✓	0	6	sum	✓	-	✓	-	-	✓	∼	∼
GLQ	✓	2	0	-	-	-	✓	-	-	✓	-	-

[1]Using data types [2]Loop perforation [3]Task skipping and memoization [4]multiple inexact program versions [5]stochastic computing
*** *very good*
** *good*
* *moderate*

3.2 Applying approximate computing

Approximate computing techniques follow the same spirit to server different use-cases. However, there are major differences. Some techniques are applied at the algorithm level, others require an exotic hardware setup. Moreover, all methodologies rely on the statistic of the input data. Variations of data, different applications, and the benchmarking setup hinder a direct and fair comparison. Quality metrics are application-specific and a *user acceptable quality* might be differently defined.

Table 3.1 presents an overview of all methods that are explained in detail below. We rank the methods based on how easily they

are supported in current software and hardware environments. Loop perforation is simple and applicable at algorithmic level and henceforth improves performance on all systems. Depending on the native support of data types and software implementation success of TSM and IPV, those approximate techniques are supported on current systems. However, stochastic computing is not supported at all and always includes customized hardware designs.

We rate the methods based on the overall applicability, generality, expected quality degenerations, and expected performance gains. We think framework based approaches such as TSM and IPV are successful on larger systems composed of many elementary kernels. However, they do not help in designing of fundamental approximation techniques for specific elementary kernels. Loop perforation is general, however, it substantially degenerates quality in some cases. Correctly using data types is adequate in most cases. Mixed precision approximation techniques rise their importance as more formats become natively supported on new hardware.

The use of datatypes

Choosing number formats in applications is a general approach that applies to all applications operating with numbers. However, we omit a numerical analysis at this stage, since we study common formats, such as IEEE 754 half, float, and double, in Chapter 5. The common formats are considered as a corner case of the introduced reduced precision formats.

Loop perforation

Loop perforation is applicable in general contexts. We identify that core loops of PageRank and GLQ are *control loops* depending on input configurations. GLQ has no remaining loops left. PageRank has two tunable loops left that iterate over the matrix-multiplication. BLSTM has six loops that stem from linear operations of the kernel.

Task skipping and memoization

The identified data parallel patterns [119] inside the kernels follow the summation-and-reduction pattern that is common to applications

relying on linear algebra. Patterns, such as stencil, scatter, or gather operations do not occur in the considered three kernels. In some cases, applying loop perforation is equivalent to identifying the typical pattern and applying task skipping. For example, studying loop perforation in summation-and-reductions occurring in dot products covers the task skipping approaches. We identified the map pattern occurring in BLSTM for activation function evaluations including sigmoid and trigonometric functions. Lookup table-based designs are applicable, however, we do not expect relevant overall performance gains since they only constitute a small part of the total workload.

Using multiple inexact program versions

The approaches described in Section 3.1.4 apply to larger systems composed of modular kernels. Since PageRank and GLQ are closed components, a further decomposition does not allow the proposed methods to be apply. BLSTM can be forced to be decomposed into three parts, computations performing the forward, backward, and CTC operations. Still, there are no further gains expected as approximation levels are already studied with loop perforation that considers all loops.

Stochastic Computing

Stochastic computing requires two aspects that limit this approximation technique. First, the underlying application needs to be demonstrated to be error-resilient and suitable for stochastic computing. Second, stochastic computing requires the development of substantial parts of specialized hardware. Those two aspects make stochastic computing interesting for *specific* use-cases but they are not suitable for general-purpose computing.

Stochastic computing demonstrated the following success. It reduced the complexity of an FPGA implementation that controls an induction motor [126]. In addition, it improves low density parity checks (LDPCs) [127, 128]. The widespread applicability of error-correcting codes in WiFi communications justifies the development of

customized hardware in this case. A narrow research branch specialized stochastic computing for neural networks. Back in 1993, gate-level simulations solved an optical character recognition problem [129]. Low-level technical improvements are discussed in detail [130] and led to solutions in 2001 [131] that used a two-layered network. Compared to a floating-point reference, the model yielded quality results within 10 percent while it achieved an order of magnitude improvement in terms of required clock cycles. In 2016 stochastic computing was applied to the MNIST dataset [132] with accuracy degenerations lower than one percent [133]. Even though these results are promising, we are not aware of recent work that applies stochastic computing on today's, state-of-the-art networks that range in the order of hundreds of layers and are exercised on more complex classification tasks.

3.3 Selected results on benchmarks

We present the results of assessing control loops and of applying loop perforation to the tunable loops of the three applications.

3.3.1 PageRank

The outer loop of PageRank is a control loop that iterates until convergence. The inner loops perform matrix-vector product like updates. The middle loop iterates over independent matrix rows and the innermost loop iterates over compressed sparse row (CSR) encoded sparse row entries. Due to the presence of the outer control loop, the approximation of inner loops changes the behavior of the outer loop. If the inner approximation degenerates results above some level, the outer control loop never meets its target and the kernel gets stuck in an infinite loop. In such cases, applying loop perforation does not work at all.

Control loop

The choice of the stopping criterion ϵ affects the number of iterations of the control loop. The total performance is directly proportional to the iteration loop count. Figure 3.1 shows the total number of

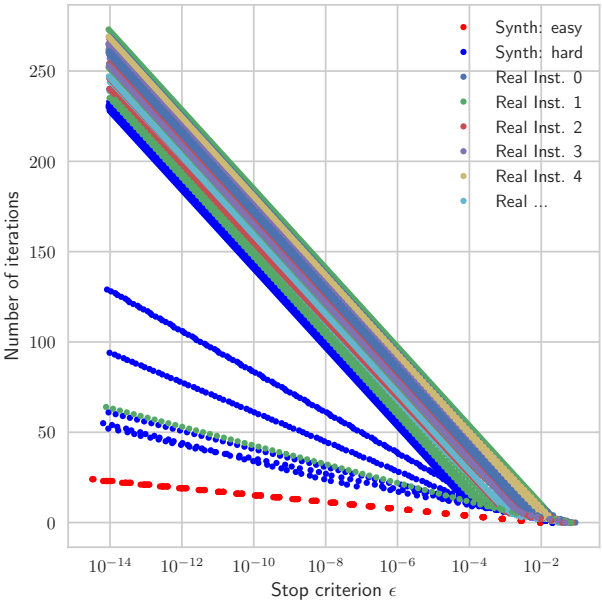


Figure 3.1: Influence of stopping criterion on number of iterations. Different problem instances obey different convergence speeds but all of them can similarly profit from relaxing the required error residual.

iterations. The number of iterations depends on the stopping criterion ϵ and the convergence speed obtained on a specific input. Changing ϵ by one order of magnitude causes the iteration count to change by a constant. Figure 3.2 shows how stricter relative stopping criteria ϵ lead to improved results measured as absolute L_2 norm of the difference between current iterations and the full precision converged reference result. To better assess the quality of the result relevant to users, Figure 3.3 shows the quality in terms of correctly ranked positions of nodes. After a minor part of the iterations, results start to stabilize and there is no need to further iterate since the ranking is no longer affected. By tuning the operation point we achieve speed-ups in the order of 4× at less than 1% point accuracy loss on the easy datasets.

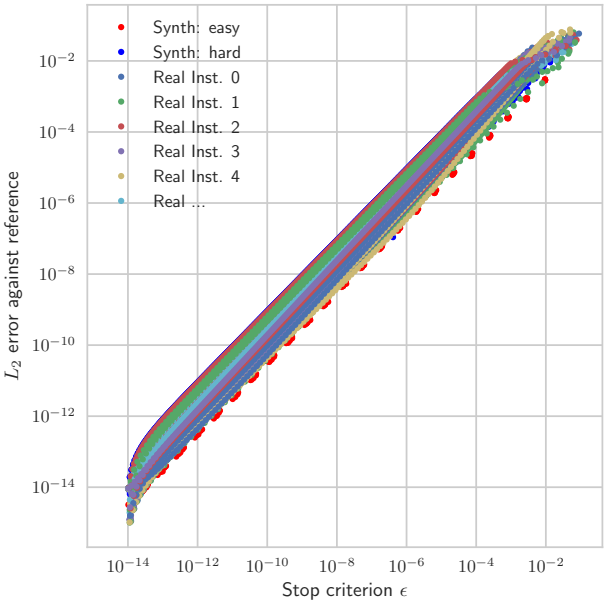


Figure 3.2: Influence of stopping criterion on achieved quality measured as L_2 norm of the difference against the converged result. In all cases, the relative residual is reduced proportionally to the absolute residual measured against a full precision implementation.

Tunable loops

First, we observed that applying loop perforation to the sum pattern in the innermost loop does not work. By construction of the problem and the observation that the iteration vector is a valid probability distribution, all involved values are positive. Henceforth, skipping entries cause the approximated sum to be smaller than the correct one. Introducing approximations that strongly biases results in one direction causes the kernel to fail to converge.

Applying loop perforation to the middle loop works if the following two criteria are met. First, since skipped iterations avoid dot product

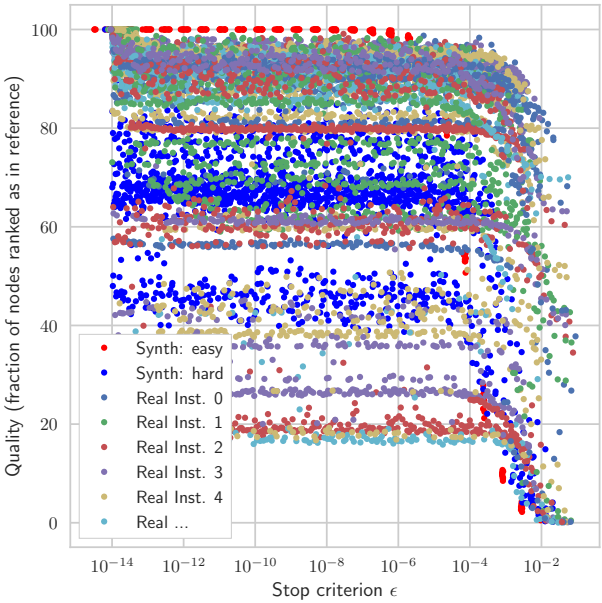


Figure 3.3: Influence of stopping criterion on achieved quality measured as percentage of ranks that equivalent as in the converged reference.

operations, we ensure to approximate the missing value with the value of the previous iteration. Second, we ensured different offsets in the approximated loops for subsequent control loop iterations. The latter point prevents generating a strictly equivalent data access pattern, that would converge to the wrong result. If convergence is reached, skipping row updates and replacing missing values with previous results becomes an error-free operation.

We run PageRank with conservative settings that impose a stopping criterion of $\epsilon = 10^{-14}$. We evaluate the algorithm on two synthetic datasets and one dataset extracted from real data. We used a random Bernoulli distribution with parameters $p_1 = 0.01$ and $p_2 = 0.001$ to generate sparse input graphs in the synthetic case.

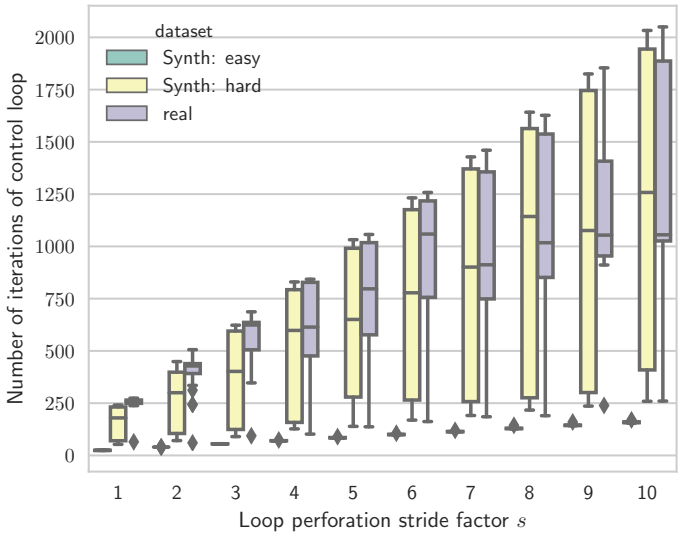


Figure 3.4: Total number of control loop iterations of PageRank. The stronger loop perforation is applied, the more additional control loop iterations are triggered.

Since loop perforation of the middle loop causes approximations of intermediate results the outer quality control loop converges slower. However, the outer quality control loop ensures achieving the same strict error reductions as in the baseline. That way, the outer control loops recover—at the cost of performance—precision that is lost due to applying loop perforation in the middle loop. Figure 3.4 shows the increased number of iterations of the control loop for the easy, hard, and real datasets. Depending on the specific input instances, we observe high fluctuations in the iteration counts. In all cases, applying loop perforation causes the control loop to iterate longer.

To assess if applying loop perforation is beneficial, we measure the total workload. Overall performance is improved if the loop perforation overcompensates the slow convergence. Loop perforation with stride factor s performs s less workload in the PageRank body.

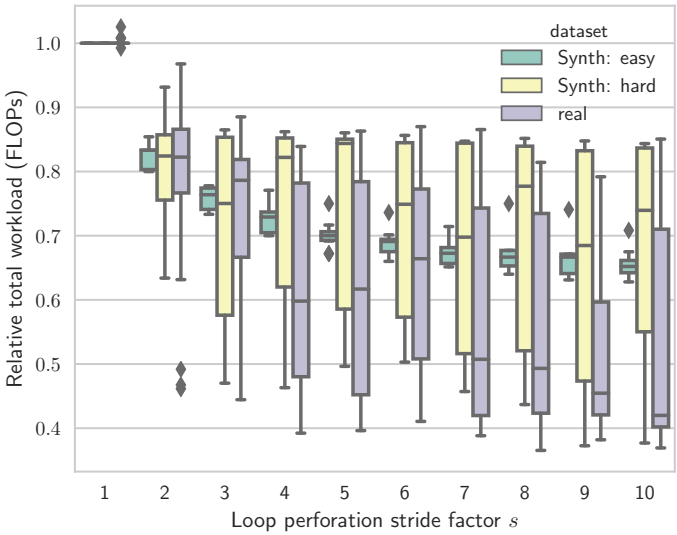


Figure 3.5: Total workload for PageRank when applying loop perforation. The local gains are large enough to overcompensate the additional control loop iterations and lead to a total reduction of the workload.

Overall, the reduction is strong enough to successfully overcompensate the additional control iterations. Figure 3.5 shows the total workload against the reference for the easy, hard and real dataset. Average gains of 35%, 25%, and 58% are achieved in terms of overall FLOP reductions without degenerating results.

3.3.2 BLSTM

We applied loop perforation to nested loops in the time relevant parts of the BLSTM kernel. Three loops occur when the LSTM forward and the backward pass is evaluated. The outermost loop iterates over columns of an input image, the middle iterates over the number of neurons related to the current computation and the innermost loop

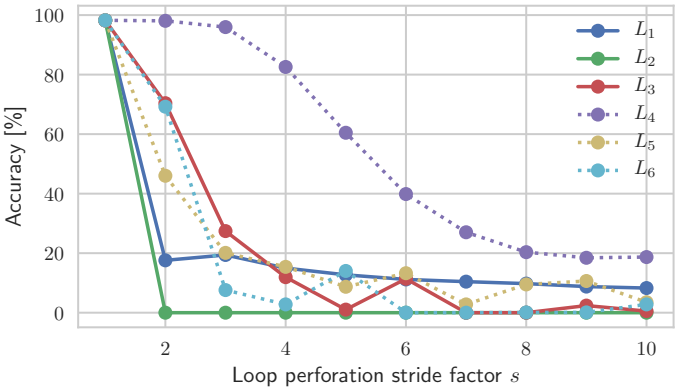


Figure 3.6: Effect on quality of BLSTM when applying loop perforation. Results quickly degenerate. Only loop L_4 can be perforated without significant degenerations.

performs a dot product operation aggregating temporary results to a single neuron. The three loops, L_1, L_2 and L_3 are perforated with strides factors ranging from one to ten. Similarly, in the CTC step, three nested loop iterations occur over image columns (L_4), the number of classes (L_5), and related summations of dot product operations (L_6). Figure 3.6 shows accuracy results when running the benchmark with different loop perforation stride factors in the range $[1, 10]$. In all experiments, only one loop was perforated at once, the other loops were operated as in the baseline (e.g., with the stride equal to one). Perforating loops causes in most cases significant accuracy drops. Accuracies below 90% are not acceptable in this benchmark. Only L_4 obeys a smoother behavior, the obtained accuracy is 98.07% and 95.97% for strides two and three that are close to the reference accuracy of 98.23%. Since L_4 is the outermost loop of three loops, the total workload of the merging block is reduced by the loop perforation stride. However, the merging block is a smaller fragment of the total benchmark that contributes in total 18.6% to the overall workload of the full kernel. Since the loop perforation of L_4 does not reduce the

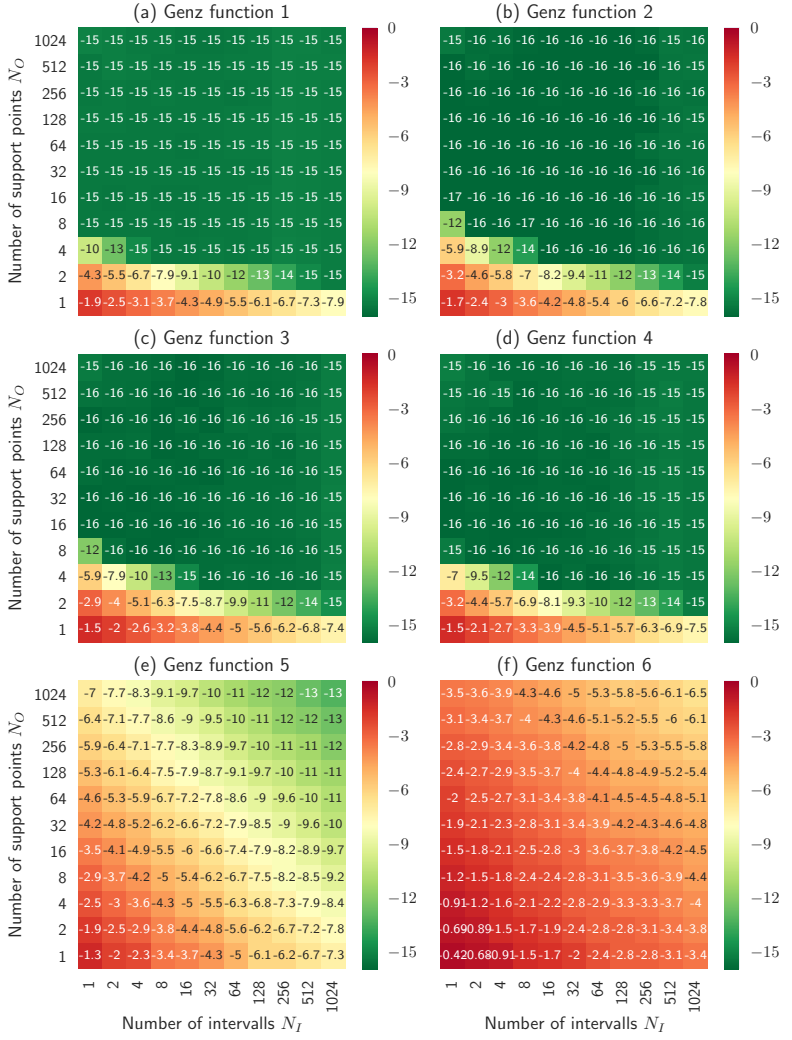


Figure 3.7: Effect of control parameters N_I and N_O on the final quality of GLQ results. Depicted are the mean of the absolute error obtained for ten runs for each of the six Genz functions. The discontinuity challenges integration routines for function 6.

heavier computation block related to the LSTM, the total performance gains are limited. Indeed, we measured a reduced run time of 4% and 5.7% of the full benchmark when running L_4 with a stride of two and three.

3.3.3 GLQ

GLQ does not have tunable loops but consists of two nested control loops. The outer loop splits the original interval into N_I partial intervals and the inner loop evaluates the quadrature by using a fixed number of support points N_O. The choice of parameters N_I and N_O strongly affects the quality of the result and determines the running time that is of complexity $O(N_I N_O)$. Figure 3.7 shows the quality obtained when sweeping independently both parameters. The quality is measured as absolute error expressed as base-ten logarithm over the average obtained when running with ten uniform sampled function parameter instances. For Genz functions 1 to 4, low orders and few interval decompositions reach machine precision error floors of about 15 to 16 digits. For example, for Genz test function 1, the configuration $N_I = 2$ and $N_O = 8$ achieves machine precision five orders of magnitude faster than a conservative configuration of $N_I = N_O = 1024$. More challenging functions, such as non-smooth functions (Genz function 5) or functions with discontinuities (Genz function 6) require higher-orders and a finer splitting to improve quality. Computing with the conservative configuration of $N_I = N_O = 1024$ and full 64-bit *double* precision the final results are up to 13 and 6 decimal digits precise for Genz function 5 and 6, respectively.

3.4 Summary and conclusion

The most important findings of this chapter are the following:

- Approximate computing techniques cover a broad range of systems, different granularity levels, and various applications. Mixed precision approaches benefit from using the smallest format that serves the needs of applications. Loop perforation improves performance by skipping iterations. Since that technique applied

to algorithms, no special hardware features are required. That makes loop perforation as generally applicable in different software stacks and deployable on various systems. Task skipping and memoization gain by subsampling or avoid triggering computations. Instead, they reuse a precomputed value of a similar input value. Using multiple inexact program versions provides interesting trade-offs for larger systems. However, this technique relies on the availability of alternative implementations for modular building blocks. Stochastic computing performs operations by using randomized bitstreams. Efficient low-cost implementations demonstrated success for specific tasks. However, stochastic computing involves customized hardware design which limits the generality and scalability of the approach.

- We evaluated the applicability of the five approximation techniques for PageRank, BLSTM, and GLQ. First, we observed that methods that are developed for larger systems, such as using multiple inexact program versions, do not provide insights on how to design the internals of elementary kernels. Second, some methods operate well for specific patterns but they might not occur in the performance-critical sections of kernels. That turns some methods limited to specific cases such as neighborhood approximation in stencil computations. We conclude that mixed-precision computing and loop perforation are simple, but the most general approaches.

- Since loop perforation is the most promising approximation technique, we reimplemented and evaluated it on the three kernels. In PageRank, loop iteration reductions in the inner loops cause longer iterations of the outer control loop. We empirically demonstrated that the reduction effect is strong than the additionally triggered loops such that loop perforation is applied with success. However, in some cases, such as BLSTM, loop perforation strongly degenerates results such that they turn the output useless.

- We identified that control loops often rely on externally provided hyper-parameters that control the operation point of the algorithm. The operation point strongly affects the quality and

performance of algorithms. Therefore, exploiting the hyper-parameter setting allows providing competitive solutions as a trade-off between quality and performance.

Chapter 4

Core transprecision concepts

In this chapter, we develop the core concepts of transprecision computing. Section 4.1 explains standard and less common representation of numerical values in a computer system. Section 4.2 abstracts a *computing system* to allow for a formal definition of transprecision computing. Within the last subsections, current design methodologies and existing solutions are identified to follow the transprecision paradigm.

4.1 Number formats

Standard number representations include fixed-point and standard floating-point representations [134]. Fixed-point representations have a long history in FPGA and ASIC design due to the simple availability of integer arithmetic inside digital signal processors (DSPs) or highly optimized designs for elementary arithmetic operations. Still, mapping general purpose applications to work with a limited dynamic range is non-trivial and requires careful considerations of the design space. In contrast, to serve the seamless design flow of complex applications in various fields, the majority of general-purpose computing processors implement the IEEE 754 floating-point standard [116] and

provide hardware support for elementary arithmetic operations for the 32-bit and 64-bit float and double-precision data types. Additional to the standard number formats, research elaborates more exotic choices of number representations such as the logarithmic number system (LNS). Even though related work addresses the design effects on algorithms [135], and implementation details of functional units [136,137], they are practically not available in the mass production portfolio of today's available processors.

4.1.1 Fixed-point

We denote with $Q_{n,f}^{unsigned}$ the unsigned and with $Q_{n,f}^{signed}$ the signed fix-point data type with a bit-width n and f fractional bits. A binary word consisting of bits $x_{n-1}x_{n-2}...x_2x_1x_0$ of length n represents the following value:

$$v_{unsigned} = \frac{1}{2^f}\sum_{i=0}^{n-1} 2^i x_i, \tag{4.1}$$

for the unsigned representation, and the same bit pattern is interpreted as:

$$v_{signed} = \frac{1}{2^f}\left(-2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i\right), \tag{4.2}$$

for signed representations. In both cases, the precision is given by $\Delta = \frac{1}{2^f}$ that defines the shortest difference between the closest representable values which is equivalently spaced over the full range. The range for unsigned and signed representations amounts to:

$$\begin{aligned} v_{unsigned} &\in \left[0, \quad 2^{n-f} - \frac{1}{2^f}\right] \\ v_{signed} &\in \left[-2^{n-f-1}, \quad 2^{n-f-1} - \frac{1}{2^f}\right]. \end{aligned} \tag{4.3}$$

Signed representations use the two's complement to map the upper half of the unsigned representations to the negative range by adding a constant value of -2^{n-1}. Since the presence of the most significant

bit x_{n-1} determines the sign of the number, it is often called the signed bit. Signed representations expose roughly half of the range of unsigned representations in terms of absolute values for a fixed bit-width, or they require an additional bit to cover the same range of positive values.

The fixed-point arithmetic follows the standard integer arithmetic if the position of the decimal point of inputs and outputs is matched. That allows to reuse the same hardware components as for regular integer arithmetic that can be understood as corner case of fixed-point arithmetic with $f = 0$. By correctly considering input and output formats addition, subtraction, and multiplication can be performed without error, by adding one bit (for a potential carry) and by adding the input and output bit-widths in the case of multiplication. For example, a 16-bit multiplier takes two 16-bit inputs and produces a 32-bit output, can perform any operation of the form $Q_{16,f_1} + Q_{16,f_2} = Q_{32,f_1+f_2}$. The availability of optimized pre-designed computing components promotes fixed-point formats as preferred candidates to implement unrolled data paths in FGPA or ASIC designs.

4.1.2 IEEE 754 floating-point

We denote with $T_{w,t}$ the IEEE 754 conform numeric representation of a floating-point value, consisting of w exponent bits and t trailing significand bits. A value v is represented by a sign s, an exponent e and the significand m as follows:

$$v_{float} \quad = \quad (-1)^s \cdot 2^e \cdot m. \tag{4.4}$$

The exponent e is stored as an integer represented with w bits and the significand m is stored as $Q^u_{t,t}$ as fixed-point number containing only fractional bits. The IEEE 754 standard [116] defines five basic formats, three binary formats and two decimal formats that are based on radix two and ten respectively. The three basic binary floating-point formats include *single*, *double*, and *quad* precision, are mapped to 32-bit, 64-bit and 128-bit representations. Most hardware natively supports the basic *single* and *double* formats. The larger *quad* formats are rarely supported on hardware and most applications fully rely either on the intrinsic *double* precision or completely switch to

Table 4.1: IEEE 754 floating-point configurations.

Name	Formula	*half* *binary16* ($T_{5,10}$)	*float* *binary32* ($T_{8,23}$)	*double* *binary64* ($T_{11,52}$)
emax	$2^{w-1}-1$	15	127	1023
emin	$1-emax$	-14	-126	-1022
Δ_{min}	2^{-t}	$9.8 \cdot 10^{-4}$	$1.2 \cdot 10^{-7}$	$2.2 \cdot 10^{-16}$
Δ_{max}^{norm}	$2-\Delta_{min}$	1.99902	1.99999988	1.999[...]
$\Delta_{max}^{sub.}$	$1-\Delta_{min}$	0.99902	0.99999988	0.999[...]
Largest	$2^{emax} * \Delta_{max}^{norm}$	$6.6 \cdot 10^{4}$	$3.4 \cdot 10^{38}$	$1.8 \cdot 10^{308}$
Smallest, norm.	2^{emin}	$6.1 \cdot 10^{-5}$	$1.2 \cdot 10^{-38}$	$2.2 \cdot 10^{-308}$
Largest, sub.	$2^{emin} * \Delta_{max}^{sub.}$	$6.1 \cdot 10^{-5}$	$1.2 \cdot 10^{-38}$	$2.2 \cdot 10^{-308}$
Smallest, sub.	$2^{emin} * \Delta_{min}$	$6.0 \cdot 10^{-8}$	$1.4 \cdot 10^{-45}$	$5.0 \cdot 10^{-324}$

numerical software libraries (that might be very slow) but in contrast support arbitrary length number representations. Even though the 16-bit *half* representation is not considered a basic format, it was introduced in the 2008 revision of the IEEE 754 standard [116]. Since recent GPUs provide full support for that described data type, we consider the binary16, binary32, and binary64 as the standard formats and we refer to the latter two with the used C/C++ keywords *float* and *double*. Table 4.1 summarizes the parametrizations and maximum absolute positive values of the standard IEEE 754 formats. Note, that the representations are fully symmetrical for negative values and are represented by setting a leading sign-bit such that the storage width amounts to $1 + w + t$ bits. The IEEE 754 standard [116] further defines special cases (not a number (NaN), Inf) and rounding behavior of arithmetic operations. Typical implementations follow a round-to-nearest with a ties-to-even rounding policy that is used as standard throughout this work.

In contrast to fixed-point, typical floating-point hardware units [138–140] are complex and operate in three stages: a) they equalize the input numbers based on the exponents, b) they performing the core operation, and c) they re-normalizing the output if required. In contrast to fixed-point formats, the operations are not exact, due

to the quantization that occurs in the final rounding step where the internal mantissa is shortened to the output significand width.

4.1.3 Logarithmic number system (LNS)

A LNS represents numbers similar as the floating-point standard but without mantissa. The number only consists of an exponent e;

$$v_{LNS} \quad = \quad (-1)^s \cdot 2^e. \tag{4.5}$$

The exponent is represented in the $Q_{n,f}^{signed}$ fix-point format with a dominant fractional part to obtain a finer granularity over the representable range. Special cases, such as NaN, Inf, and Zero are encoded with specific bit patterns. Table 4.2 summarizes the maximum positive values that cover similar ranges as for their standard floating-point equivalent types. We denote the LNS format by $L_{n,f}$ where the parameters n and f denote the bit-width and the number of fractional bits used in the underlying fixed-point exponent representation. The LNS has gain research attraction already in the 1970's by the observation that some operators massively simplify through the setup of the number system [141, 142]. For example, integer addition, integer subtraction, and bitshift operations can compute multiplications, divisions, and square-roots of LNS numbers: Let $v_1 = (-1)^{s_1} \cdot 2^{e_1}$ and $v_2 = (-1)^{s_2} \cdot 2^{e_2}$ denote the LNS representation of two values v_1 and v_2. Then, Equation (4.6) shows how operations map to the underling representation. Modified arithmetic logic unit (ALU)'s with special-case handling allow power, area-efficient, and fast implementations of those operations.

$$\begin{aligned} v_1 \cdot v_2 &= (-1)^{s_1+s_2} \cdot 2^{e_1+e_2} \\ v_1/v_2 &= (-1)^{s_1+s_2} \cdot 2^{e_1-e_2} \\ \sqrt{|v_1|} &= (2^{e_1})^{0.5} = 2^{0.5e_1} \end{aligned} \tag{4.6}$$

In contrast, addition and subtraction involves the evaluation of non-linear functions as follows:

$$\begin{aligned} v_3 &= v_1 \pm v_2 \\ e_3 &= \max(e_1, e_2) + \log_2(1 \pm 2^{-|e_1-e_2|}). \end{aligned} \tag{4.7}$$

Table 4.2: LNS configurations similar to *half* and *float* data types.

Name	Formula	16-bit $L_{15,10}$	32-bit $L_{31,23}$)
emax	$2^{n-f-1} - \frac{1}{2^f}$	$16 - 2^{-10}$	$128 - 2^{-23}$
emin	$-2^{n-f-1} + \frac{1}{2^f}$	$-16 + 2^{-10}$	$-128 + 2^{-23}$
Largest	2^{emax}	$6.5 \cdot 10^4$	$3.4 \cdot 10^{38}$
Smallest	2^{emin}	$1.5 \cdot 10^{-5}$	$2.9 \cdot 10^{-39}$

LNS research focuses on the two non-linear functions $F^+(r) = \log_2(1+2^{-r})$ and $F^-(r) = \log_2(1-2^{-r})$ with $r = |e_1 - e_2|$ that build the core operation of addition and subtraction units. All hardware implementations are based on lookup and interpolation methods [135]. Since $F^-(r)$ has a singularity for $r \to 0$, critical intervals of r close to zero are decomposed with so called *cotransformations* into functions that are simpler to approximate [143–145]. Authors of early papers concluded that 12-bit width LNS addition/subtraction units are infeasible to the exponential requirements of the lookup tables. Recent implementations demonstrated the feasibility of 32-bit LNS units that deliver roughly the same dynamic range and precision as the IEEE 754 32-bit *float* implementations [71, 146]. However, due to the exponential scaling of lookup tables for higher precisions, 64-bit LNS formats are infeasible to be designed in a competitive manner compared against the traditional IEEE 754 64-bit *double* implementations.

4.2 The transprecision system view

Before we discuss transprecision methodologies, we firstly define an abstract definition of an arbitrary computing system. We require a general, yet simple enough, description to classify existing work and to address complex interacting tasks that cover all aspects from physical foundations up to algorithmic implementations. To define an abstract computing system, we first provide a list of aims we want to cover with the abstraction.

- The abstraction should be general to cover any existing computing system.
- The abstraction should be extensible to cover new computing systems.
- The abstraction should be used to define and explain the methodologies of transprecision computing.
- The abstraction should allow deriving methodologies needed for transprecision computing.

We define three key concepts, *components*, *interfaces*, and *actions*:

Component. Identifies parts of a computing system. Components can be encapsulated or replicated. Components express structure and help to identify the granularity level of considerations.

Interface. Identifies relations among components by defining expected inputs and outputs.

Action. Explains events that occur. Actions might be triggered by other actions, receive input, and create output.

The three concepts are defined in a sufficiently general fashion such that terms component, interface, and action can refer to both software and hardware.

Definition 1 *(Abstract Computing System). We define an abstract computing system consisting of two particular components: a) a* descriptive component *and b) a* physical component *where relations are defined through* interfaces*.* Actions *define the purpose of the system.*
Synonyms: Personal computer, microcontrollers, serves, dedicated systems including ASICs and FPGAs.
Typical: IBM Power8/9, NVIDIA GPU K80, P100, V100.

The above definition splits a computing system into two specific parts that we assume are always present, a descriptive part that defines or implements actions and a physical system that can process information and solve a task. This notation fits classical computing systems, FPGA workflows, and ASIC designs.

Classical computing system. Physical components define the traditional hardware, including the processor, the cache memory hierarchy, the DRAM, hard drive, and peripherals. Descriptive components are defined through the classic software parts that define the

behavior of the classical computing system, including the operating system, software libraries, and specific implementations.

FPGA. Physical components refer to the specific board or subcomponents, such as cells, DSPs, or specific circuits. Descriptive parts refer to hardware description language (HDL) abstractions that define the behavior of the board once configured.

ASIC. Physical components refer to the manufactured integrated circuits and subcomponents thereof and descriptive parts, similar as for FPGAs refer to source files that describe the hardware.

To further abstract a computing system, we use the key concepts of *micro*, *macro*, and *system* levels to distinguish different granularity considerations of a system. The terms *micro*, *macro*, and *system* relate to different components depending on the focus of consideration. For example, to study a compiler, the terms *micro*, *macro*, and *system-level* refer to assembly code, single-line statements, and function routines respectively. However, in a large software project, the same terms refer to function routines, libraries, and applications.

Definition 2 *(Matching work into the abstraction). To understand novel work, we match it against the definition of a computing system and identify three concepts:*
a) Structure: by identifying (1) components, (2) interfaces, and (3) actions.
b) Granularity: by ranking components into: (1) micro, (2) macro, and (3) system levels.
c) Importance: by classifying components into: (1) core components, (2) tooling components, and (3) side components.

Definition 2 builds a guideline on how related work can be classified. Structure and granularity define the scope of the work. Additionally, components are classified according to their importance. Core components define the focus of the work, tooling components refer to components that were used during development, but their internal operation is out of the scope. Side components note components that are inherently assumed to be present but out of the focus of the work. For illustration, in a classical software project the core component refers to the source code, tooling components refer to the compiler and

the profiler, and the side component refers to the used hardware. In contrast, the core component of a classic ASIC design is the hardware design of a unit. Tooling includes the full stack of specialized software products to perform compilation, estimation, place and route, and validation of designs. Side components of an ASIC work reefers to units present in the final system that are not part of the current design considerations.

The abstract view enables us to quickly characterize existing work by identifying the structure, the granularity, and the importance of descriptive and physical components. The abstraction is general and extendable, as required by the first two statements at the beginning of this section. We use the conceptual idea to introduce transprecision computing and related methodologies in the next section.

4.2.1 Transprecision concepts

To demonstrate transprecision computing as a general concept that improves the next generation of computing systems, we establish key conceptual metrics. They assess different solutions of computing systems to solve a given task. We define quality and performance as follows:

Definition 3 *(Quality). Quantifies a measurable, reproducible, success of a solution obtained with a computing system to a given task. The quality is solely a property of the obtained result.*
Synonyms: Accuracy, Algorithmic-Performance, Result-Performance.
Antonyms: Result-costs, Residual, Error.
Typical: Top-k accuracy, L1 or L2 residual error.

Definition 4 *(Performance). Quantifies a measurable, reproducible, amount of work accomplished by a computing system to achieve any solution to a given task. The performance is solely a property of the system and how it behaves to obtain a solution.*
Synonyms: Execution speed, throughput, bandwidth.
Antonyms: Low latency, short time, low complexity, low power.
Typical: Runtime measurements, Energy-to-solution measurements.

The provided definitions allow the output to be quantitatively assessed independently from the process that generated the output.

In some literature, the terms quality and performance are defined differently to express the same statements. To avoid disambiguation, consider the statement *[...] algorithm A outperforms algorithm B in all cases [...].* If the statement is in the context that the produced results of A are better than results produced with B, it refers to *quality.* However, if the statement means that A finishes about x times faster than B, it refers to *performance* that characterizes the process that produced the result. With the definition of quality and performance we are able to state the following transprecision computing definition:

Definition 5 *(Transprecision Computing). A transprecision computing systems is a computing system characterized through quality, performance, and trade-offs there-on* $S := (Q, P, R_{P,Q})$*. The configuration space* Θ *defines transprecision root-cause settings and the transprecision system is defined through a quality characterization* $Q(.) : \Theta \to \mathbb{R}, \theta \mapsto q$*, a performance characterization* $P(.) : \Theta \to \mathbb{R}, \theta \mapsto p$ *and the trade-off thereon* $R_{P,Q} : \Theta \to \mathbb{R}^2, \theta \mapsto (p, q)_\theta$ *for* $p = P(\theta)$ *and* $q = Q(\theta)$*.*

A transprecision system based on the configuration space Θ is a family of systems that are based on the transprecision specific setting θ that affects both, the quality of the results it produces and the performance of the process that produces the results.

A transprecision system is a parametrizable family of systems. Specific settings θ in the transprecision configuration space Θ affect both, the quality and performance of the final solution. The characterization functions Q and P map the configuration θ onto scalar quality and performance metrics. Trade-offs $T_{P,Q}$ between quality and performance are obtained by stating those for an operating point specific $\theta \in \Theta$ setting. The root-cause is the effect that causes different configurations to impact quality and performance. We define a transprecision system general including various effects that act as root-causes. The generality causes some of the defined concepts to overlap with established approximate computing approaches. Henceforth, we define a list of features that distinguish a system as trans-precise:

- The configuration space Θ is non-trivial.
- Either, the *quality* or *performance* characterization functions (P and Q) are non-trivial.

- Either, the evaluation of the *quality* or *performance* characterization function span multiple *granularity levels* of a computing system.

- Either, the main effects of the *quality* and the *performance* characterization are caused by different *components* of a computing system.

The first criterion excludes cases, where for example only two algorithms are compared such as comparing a reference with an approximative formulation. The 2nd and 3rd criterion are satisfied if the root-cause effect propagates non-trivially through different granularity levels of a computing system. For example, adjusting the precision of a variable meets those requirements. Micro effects introduce a scalar quantization error, macro effects cause software library routines to produce different numerical results, and system effects impact the final application quality. The 4th criterion highlights that effects on the quality and precision of a system propagate through different components of a computing system. For example, loop perforation satisfies the first three criteria. However, it fails on the fourth criterion since performance gains and quality degenerations are caused inside the same component, namely, in the software component that defines the loop under consideration. In contrast, using reduced precision typically affects quality through the software-stack while performance is gained through the hardware-stack.

Next, we define common tasks related to transprecision systems:

Configuration space design Defines the transprecision configuration space Θ. Especially, if the root-cause impacts on micro level, typical configurations at system level are composed of many configurations, defining large configuration spaces.

Characterization Evaluates *quality* and *performance* for different configurations. Depending on the root-cause, the analysis is performed analytically, by estimation, with statistical models, or by empirically emulating the transprecision system.

Configuration optimization Deals with finding best configurations $\theta \in \Theta$. The following equation states the optimal preforming solution for a given quality constraint,

$$\theta_{opt1} = \arg\max_{\theta \in \Theta} P(\theta) \quad s.t. \quad Q(\theta) \geq q_{ref} - \Delta q, \tag{4.8}$$

where the quality constraint is expressed as maximal allowed degeneration Δq from a known reference solution q_{ref}. Similarly, the following equation states the optimal quality solution that achieves a fixed speed-up η:

$$\theta_{opt2} = \arg\max_{\theta \in \Theta} Q(\theta) \quad s.t. \quad P(\theta) \leq \eta p_{ref}. \tag{4.9}$$

A common task for transprecision systems is to find the Pareto-optimal frontier [147] of configurations between quality versus performance trade-offs. A configuration θ^1 dominates another configuration θ^2 if, and only if, is better in both aspects, the quality and performance delivered by that solution. We denote that θ^1 is the preferred solution by $T_{p,q}(\theta^1) \succ T_{p,q}(\theta^2)$. Formally, the Pareto optimal frontier is given as a set of all configurations for which no better configuration exists:

$$\Theta_{opt} = \{\theta \in \Theta : \{\theta^* \in \Theta : T_{p,q}(\theta^*) \succ T_{p,q}(\theta)\} = \emptyset\}. \tag{4.10}$$

4.2.2 Reduced precision as root-cause

Definition 5 defines transprecision computing as a system that performs tasks with a configuration dependent quality and performance. However, the root-causes that affect quality and performance are not specified in the abstraction to cover cases that obey the same philosophy. Well suited root-causes for transprecision computing satisfy the following criteria:

- Are applicable at different components of a computing system;
- Have the potential for performance gains;
- Allow defining clear interfaces between components for modular developments;
- Allow to well-define and to search configuration spaces;

- Allow to automatize the finding of good configurations.

The number format representations and their arithmetic deliver a well suited transprecision root-cause to enhance computing systems. We formalize the idea of reduced precision by assuming a number representation and arithmetic denoted as data type T parametrizable with parameters θ. Integrating reduced precision into applications and hardware causes the following chained effects:

First, quality is affected by the numeric representation and how variations propagate through an application. We denote quality characterizations caused by reduced precision as the numerical behavior of an application. The numerical behavior directly propagates through algorithmic granularity levels. On the *micro-level*, different data types cause quantization effects such as differently representing scalar values. On the *macro-level*, *micro* effects are applied to larger data chunks or chains of arithmetic computations. They accumulate and propagate quantization effects through a system.

Second, reducing the precision allows operating the hardware with higher performance, as it has to perform less work per data item. We assume that the reduced precision type T has a smaller bit-width representation than the data type used in the original implementation. The reduction has several implications and allows improving multiple hardware components. First, if the hardware supports transprecision computing features, performance gains are directly measurable. However, in some cases, only a subset of all potential configurations might be supported on hardware. That restriction might limit performance gains that would be achievable with finer granularity considerations. For example, most traditional computing cores include floating-point support for 32-bit and 64-bit IEEE 754 number representations. Recently, especially in GPUs, the IEEE 754 half type with a bit-width of 16-bits is natively supported and extends the granularity level of available precisions. Second, the feasibility of better performing hardware is well-motivated. For example, four-way vectorization similarly extends the instruction set leading to performance gain as when moving from a single 32-bit format to a two-way vectorized operation of 16-bit representations. Additionally, bandwidth-limited, contiguous memory accesses scale proportionally to the transferred data volume. Reductions in data-types translate to reduce the overall

data volume. The critical target for developers is to demonstrate that the application delivers the required quality with reduced precision computations.

Reducing precision implements transprecision concepts in a modular way. Defining the reduced precision representations and arithmetic operations, the transprecision root-cause is specified and quantified. The scalar-level definition of the data type and elementary operations build the modular *micro-level* abstraction of reduced precision computing. Software libraries extend and emulate the numeric behavior on *macro* and *system-level* of applications. The scalar-level abstractions of number format representations act as specifications for designing reduced precision units in FPGAs or ASICs.

The availability of a few primitives of non-standard data types allows analyzing complex numerical behavior on application-level without direct concerns of HW implementations. The common use of reduced precision emulation libraries implies the transprecision configuration space Θ. Emulations allow to empirically evaluate the quality of applications $Q(\theta)$. Analyzing numerical behaviour powers the development and optimization of specific problem instances.

4.2.3 Transprecision computing in current solutions

Due to the general formulation of transprecision computing, we discuss to what extent it overlaps with existing products or implementations. Additionally, we highlight the key features that we envision for transprecision computing covering the next decade. First, we observe the following:

Observation 1 *(FPGAs and ASICs). Well engineered FPGA and ASIC designs follow the concept of transprecision computing.*

A typical hardware design flow includes transprecision concepts at different stages: a) at gate-level synthesis, b) by categorical design choices, c) at register-transfer level (RTL) data path design.

First, we understand the synthesis of gate-level netlists from HDL abstractions as a corner case of transprecision computing. Proficient commercial tools, such as the Synopsis Design Compiler[1], are highly

[1] Available at `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html` (November 2019)

specialized to automatize this process. The configuration space Θ as in our transprecision formulation maps to the space of options and flags that can be passed to the synthesis process. The standard analysis of hardware designs states costs in terms of silicon area against the worst-case propagation delay. The concept of the so-called AT-plots maps to transprecision concepts by identifying the area and delay as two orthogonal performance metrics $P_{area}(\theta)$ and $P_{delay}(\theta)$. Since valid synthesized implementations obey a defined behavior, the quality is fixed. Results produced by the circuit are equivalent.

Second, designs include the opportunity for categorical choices of subcomponents or architectural choices of functional units. For example, using alternative Design Ware components affects performance metrics, e.g., choosing an 8-bit ripple-carry adder versus an 8-bit carry-lookahead adder. Quality and performance are affected when using different adders variants, e.g., a 10-bit adder instead of an 8-bit adder. High-level architectural choices, for example, sequential operation or binary-tree like reductions might impact quality as well. For example, quantization caused by rounding leads to non-commutative behavior of summations.

Third, data-path design involves defining the bit-widths of computational units and related registers that directly impact the quality and performance. Designs that rely on look-up table s (LUTs) highly profit from slim input address spaces due to the exponential area scaling. The choices for the LUT design affects the quality and performance of the final solution.

Hardware designs provide highly-performant implementations, very tiny designs, and low-power operation. Those solutions are obtained by investments including long development time, human expert knowledge, and engineering, licensing, and manufacturing expenses. Driven by those factors, engineers take decisions to develop and achieve reasonable solutions. Depending on the effort spent to make design space elaborations including numerical aspects, such implementations follow the transprecision computing paradigm.

However, most designs rely on expert knowledge to make reasonable assumptions. They are missing the systematic elaboration of the numerical configuration space with functions describing effects on quality and performance. Moreover, most design questions are considered for specific circuits, which impedes to generalize and reuse

similar considerations for new problems. In our vision, transprecision computing scales as a general concept that deploys efficiently to new settings.

Observation 2 *(Vision of transprecision (TP)). Transprecision computing aims to co-design hardware and software for general-purpose systems. The provided solutions share the flexibility known from general-purpose computing but aim to deliver solutions that obey performance gains known from ASIC development.*

In this chapter, we have defined and abstracted the paradigm of transprecision computing. The definition is as general such it covers aspects from physics, hardware design, up to the regular system and application development. The long term vision includes a view that delivers a computing system as easy-to-use as known from software development. At the same time, transprecision computing delivers performance superior to today's available hardware. In contrast to regular programming, the achieved performance gains stem from quality versus precision considerations that affect multiple software and hardware components. Those effects materialize with performance improvements that are similarly known from FPGA or ASIC design flows. Since the full extent of this vision is out of the scope of this thesis, we focus on the core components. To that end, Chapter 5 presents the implementation of a reduced precision library, followed with the application level integration of deep learning use-cases presented in Chapter 7.

4.3 Summary and conclusion

The most important findings of this chapter are the following:

- We present the two predominant number representations, the two's complement fixed-point representation, and the IEEE 754 floating-point standard. We highlight the importance of the availability of high dynamic range computations that justify the use of floating-point formats. We summarized the LNS as an alternative to the IEEE 754 standard that delivers similar dynamic ranges and precision levels up to 32-bit implementations.

Research around LNS focuses on the implementation of units for addition and subtraction that are inherently based on lookup and interpolation methods.

- We define a *computing system* consisting of descriptive and physical components. Further, we define a transprecision computing system as a configurable system that produces results with measurable performance and they achieve a measurable quality. We identify the concept of reduced precision in numeric representations as root-cause for transprecision computing. Developing and evaluating a transprecision system involves the following steps:

  1. Identify the transprecision root-cause.
  2. Abstract impact of root-cause on application quality.
  3. Abstract impact of root-cause on processing performance.
  4. Evaluate the transprecision system by implementing, emulating, or estimating core components.
  5. Define and solve specific goals. (Design configuration space, use configuration search algorithms, solve for specific quality constraints).

- We observe that the concept of transprecision computing is already applied with success for certain FPGA or ASIC design flows that include systematic numerical considerations to improve performance. We envision that future general-purpose systems can profit from similar considerations.

Chapter 5

Emulating numerical behavior of applications

The conventional IEEE 754 single- and double-precision floating-point formats dominate current hardware architectures where a few recent GPUs extend full support for arithmetic operations to the IEEE 754 half format. The discussion in Section 4.2.2 suggests revealing the hidden benefits of customized FPUs that operate with non-conventional formats at a reduced precision. In particular, such considerations are especially important in applications that justify the development of ASICs of FPGA designs. However, since traditional software is written for classic processors, most of the existing code performs numeric computations with natively supported data types only. While engineering work for ASIC and FPGA designs includes numerical studies, they are mostly performed in application-specific settings for conventional fixed-point formats that are directly supported on the system for which they are developed. To demonstrate the generality, feasibility, and scalability of transprecision computing, it becomes necessary to allow for an simple software integration. Numerical behavior of non-conventional reduced precision number formats is required to be studied at the application level. To that end, we present a flexible C++ emulation framework, named floatx, to investigate the effect of reduced-precision floating-point formats in numerical

applications. Here, reduced or low precision refers to a data type that does not have more significant or exponent bits than IEEE 754 double precision-format.

5.1 The floatx library

5.1.1 Related work

Many software packages exist that allow operating numeric computations with non-native floating-point precision. Among these, the GNU's not unix!; free software (GNU) multiple precision floating-point reliably (MPFR) [148] library[1] is a de-facto standard for arbitrary higher than regular precision, used for example in gcc. Similarly to GNU MPFR many other floating-point emulation tools (see, e.g., the survey in `https://www.mpfr.org/`) provide software support for extended precision and, therefore, serve a different purpose. In the following review, we target floating-point emulation tools that focus on reduced floating-point precision.

INTerval LABoratory (INTLAB) [149] is a Matlab package that offers the fl-numbers, a concept similar to floatx. Fl-numbers have at most 26 significand bits (including the implied one); the maximal exponent range is $[-241, 242]$ (both lower than that in floatx numbers); and global variables exist that control the numbers of significand and exponent bits in effect[2]. That last trait makes INTLAB slightly less flexible than floatx, where the precision is a property of the data itself.

The Sipe [150] C library[3] is a header-only tool for experimenting with floating-point algorithms and very low precision. It supports the correctly-rounded basic arithmetic operations, except divisions and square roots, but with fused-multiply-adds (FMAs). Compared with floatx, it stores the number's value either in a native backend floating-point type or as a pair of two integers (for the non-normalized significand and the exponent parts, respectively), while the precision is a property of an operation on the data.

[1] Available at `http://www.mpfr.org/` (version 4.0.1, February 2018)

[2] Available at `http://www.ti3.tuhh.de/intlab/demos/html/dfl.html` (Accessed September 2019)

[3] Available at `http://www.vinc17.net/research/sipe/` (Accessed September 2019)

Precimonious [151] is a program analysis tool based on low level virtual machine (LLVM) that analyses floating-point program variables in an attempt to lower their precision. This tool recommends the smallest data type for each variable that produces an accurate enough answer for a representative set of program inputs.

Universal numbers (UNUMs) and their new versions, posits, [152] are variable-size alternatives to the IEEE 754 formats. The floatx library also provides variable-size formats but those are based in the IEEE 754 standard and the number of bits is fixed prior to any computations. In contrast, the unum formats cloud grow automatically if required by the computation. Furthermore, there is no easy mapping between the unum and IEEE 754 formats.

Berkeley's SoftFloat [153] is a library that provides a floating point IEEE 754 implementation using only integer operations. This implementation is limited to the standard types plus the legacy 80-bit format from Intel/Motorola. In contrast, floatx uses the floating-point hardware to greatly reduce code size with respect to SoftFloat as well as to support any format smaller than the underlying base format. Also, floatx leverages C++ overloaded operators to ease integration in existing programs.

FlexFloat [154] is a C software library enhanced with C++ wrappers that enables exploration of numerical effects by tuning both precision and dynamic range of program variables. The purpose of FlexFloat is, therefore, very similar to that of our floatx. Nonetheless, as we discuss in the following section, floatx presents several programming advantages, due to the adoption of C++ as the backend framework language, that are difficult to attain with a solution based on C.

5.1.2 Interface and design goals

The floatx library was developed to simplify the transition from existing code to reduced precision. To that end, it is designed to follow an intuitive and easy-to-use interface whenever possible and it implements the core functionality of emulating reduced precision floating-point formats that are coherent with the IEEE 754 standard. The library is designed around the following goals:

- The floatx types should be an extension of the standard binary floating-point types (float and double), and their interface should be equivalent to them. This means that, for any two floatx objects, a and b, the following expressions/operations should offer the expected semantics, as in the case of float and double:

  $$a + b, a/b, ...$$

  $$a < b, a >= b, ...$$

  $$a = b, a+ = b, ...$$

- The floatx types should also be interoperable with built-in numeric types (signed and unsigned integers, built-in floating-point types) — the expressions above should be valid even if a and b are different floatx types, or if one of them is a built-in type. This implies that floatx types should support a set of implicit conversions compatible with standard numeric promotions and numeric conversions of built-in types.

- The size of a floatx object should never be larger than the size of a double. This simplifies the porting of existing codes to operate on top of floatx, as it is then possible to embed a floatx value into the storage space originally used for a double. This ensures that parts of the code which just move or read data, but perform no floating-point operations, do not have to be modified.

Listing 5.1 provides a small example that demonstrates the features and interface of floatx.

Lines 1, 2, and 7 show how floatx numbers can be constructed from built-in types (floating-point numbers and integers) and read from C++ streams. Lines 8 and 9 show how these objects are used to perform basic arithmetic and relational operations. Lines 10-13 demonstrate the interoperability between different floatx and built-in types. The comments on the right specify the return type of the operation. Note, that T == U, where T and U are types, is used to convey that these two types are the same, i.e., that std::is_same<T, U>::value evaluates to true. Lines 8 and 11-13 also show that floatx types can be implicitly converted to other floatx types or built-in

```
1  flx::floatx<7, 12> a = 1.2; // 7 exponent bits, 12 sign. bits
2  flx::floatx<7, 12> b = 3;   // 7 exponent bits, 12 sign. bits
3  flx::floatx<10, 9> c;       // 10 exponent bits, 9 sign. bits
4  float              d = 3.2;
5  double             e = 5.2;
6
7  std::cin >> c;
8  c        = a + b;          // decltype(a + b) == floatx<7, 12>
9  bool t   = a < b;
10 a       += c;
11 d        = a / c;          // decltype(a / c) == floatx<10, 12>
12 e        = c - d;          // decltype(c - d) == floatx<10, 23>
13 c        = a * e;          // decltype(a * e) == floatx<11, 52>
14 std::cout << c;
```

Listing 5.1: Sample code using floatx.

types. Finally, line 14 shows how floatx types can be written to an output stream.

5.1.3 The choice of C++

Given the restrictions imposed by the design goals stated at the beginning of this section, the language of choice needs to support a powerful data type system, which enables programmers to define their own types and operators on those types. Besides, the language has to support some sort of type arithmetic, which is general enough to specify numeric promotion and implicit conversion rules for custom types. Furthermore, the design goal of supporting custom types that are almost as precise as double and fit then into the same storage space as double, means that custom types are not allowed to incur any memory overhead. The information about the precision of the type cannot be maintained as additional data and has to become a part of the type itself. These requirements discard C as a possible candidate. In contrast, C++ programming language supports both user-defined types and operator overloading, and the abstractions built in C++ do not impose any memory overhead. C++ templates provide an easy means to incorporate precision information into the type as well as to define arbitrary conversion rules using this information. The latter is possible since C++ templates (are believed to) form a Turing-complete language [155], evaluated during compilation. Additionally, most C++ compilers are capable of inlining function calls and

optimizing output expressions that can be evaluated at compile-time, which improves the performance of floatx types.

Applications using C and Fortran 77 can be ported to C++ (and thus, integrated with floatx) with a reasonable programming effort. Since C++ maintains a good level of compatibility with C, applications written in the latter language can usually be compiled as C++ applications with minimal modifications. Fortran 77 applications (unlike more modern Fortran variants) can be translated into C using the tool f2c and then compiled with a C++ compiler. In Section 6 we detail how we integrate floatx behind PyTorch [62] to provide the functionality in a Python-based deep learning framework to ease the integration in any deep learning model.

5.1.4 The floatx class template

To achieve the third design goal specified in the introduction of this section, and to satisfy the first and second goals, different precisions in floatx are implemented as distinct specializations of the floatx class template. The number of exponent and significand bits used in the format are encoded into the type via integral template parameters.

The floatx library uses a hardware-supported floating-point type as backend, which in turn is used to store the data, simplify the implementation, and improve the performance of arithmetic and relational operations. The binary representation of any floatx value is equivalent to the rounded and (whenever possible) normalized representation of the same value in the backend type. Note that the set of all representable values in a backend floating-point type T_{w_B,t_B}, with w_B significand bits and t_B exponent bits, is a superset of all representable values in any floating-point type with $w \leq w_B$ significand bits and $t \leq t_B$ exponent bits. Thus, T_{w_B,t_B} can be used as back end for any floatx type that is, at most, as precise as T_{w_B,t_B}. There are advantages and disadvantages of this approach.

On the positive side, a comparison of two floatx values amounts to a comparison of the underlying backend values, which yields a considerable performance benefit. The textual output is likewise straightforward. Additionally, all arithmetic operations are performed on those backend values. The result is then rounded following the precision of the floatx type and again stored (exactly) in the backend type.

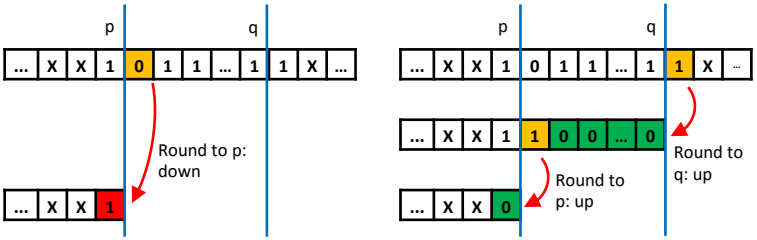


Figure 5.1: Effect of double rounding might cause different results. Left: directly rounding to a precision of p mantissa bits truncates due the 0 at position $p+1$. Right: double rounding causes twice an addition due a carry-bit in the intermediate representation.

Values such as $\pm$inf or NaN are directly stored in the backend type without any conversion. For efficiency, floating-point status flags are not supported by floatx. Also, floatx multiplication and addition are not fused in the current implementation, as this optimization is rarely worth the extra template meta programming effort.

All arithmetic operations are performed in the back-end type, using the corresponding machine's floating-point arithmetic instruction. The result of any operation is inevitably rounded back to that type, using the machine's rounding mode in effect (usually round-to-nearest, ties-to-even), before it is rounded again to the target floatx type, what is called double rounding [156, 157]

Double rounding could fail when the first step results in a value that is just at the same distance from two values in the reduced precision. In the following discussion X stands for an arbitrary bit and arrows indicated roundings. Figure 5.1 shows one example of double rounding when the bit addition carries up to the last bit before the reduced significand.

An additional example of double rounding failure is when the original value is close to the middle point in high precision as depicted in Figure 5.2. In this case, at least one bit after $q+1$ must be set in the original value. The same situation arises if the last bits after q in the original value are exactly 10...0. In both examples the error using the double rounded value is $|x - Q_p(Q_q(x))| \leq \Delta_p + \Delta_q$, which

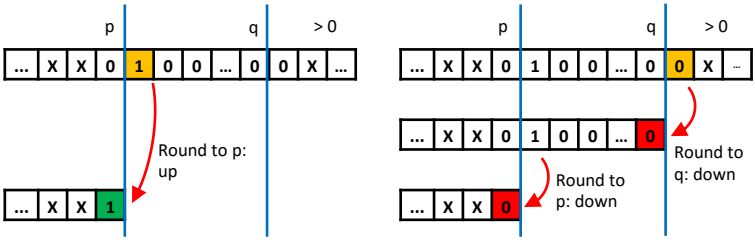


Figure 5.2: Effect of double rounding might cause different results. Left: directly rounding to a precision of p mantissa bits rounds up due to the non-zero tail of the high-precision value. Right: double rounding causes twice a truncation since the non-zero part is lost in the first rounding step.

is approximately the expected error for a significand of p bits, that is Δ_p, when $p \ll q$. This should be the typical case for floatx as, in most cases, we expect the user is interested in formats that are smaller than single precision. Formats close to double precision (more than 44 significand bits) are not quite as interesting because they will provide much lower benefits from the hardware point of view.

The correct result without double rounding can be obtained by exploiting only backend arithmetic [158]. Also, the iterative correction proposed in [159] could be used for any elementary mathematical function. The floatx library does not employ either approach because both have a large overhead not worth the extra accuracy. Another way to avoid double rounding is to employ round-to-odd [160]. This rounding mode is very efficient and without bias, but the maximum error is the same as with truncation, and twice as large as for round-to-nearest. The floatx library does not implement round-to-odd because it is not in the IEEE 754 standard. The rounding routine currently integrated into floatx has a limitation in that it only supports double as the backend type, and only the *round to nearest, ties to even* rounding mode. A more general rounding routine with support for different rounding modes and backend types is planned in the future.

```
template <int exp_bits, int sig_bits, typename backend_float = double>
class floatx {
private:
  backend_float data;
};
```

Listing 5.2: Basic structure of `floatx`.

Listing 5.2 shows the outline of the floatx class template and demonstrates how the desired binary representation is achieved. Listing 5.3 is a verbatim copy of floatx's top-level rounding routine, which demonstrates the steps required to transform a value of the backend type into a value of floatx type.

The rounding process constructs the new value by converting the exponent and mantissa from the original value (the sign is unchanged). No conversion is required for Infinity and NaN values, nor if the floatx type is configured with the same parameters as the backend type. The first step in the rounding process is to extract the bits for the mantissa, exponent, and sign using bitwise operations. This process is more complicated for the exponent because of the bias. Next, each part of the number is fitted into the target format. If the exponent is too small, the number will be denormalized; if it is too large, the result will be Infinity. The mantissa is rounded to the nearest value and corrected if it is too large. Finally, the rounded value is constructed from the original sign and the revised mantissa and exponent.

5.1.5 Operations on floatx objects

As already noted in Section 5.1.4, all operations on floatx values are trivially implemented in terms of operations on the backend data type and the rounding routine. In consequence:

- Comparing two floatx values is equivalent to comparing their representations in the backend type.
- Any arithmetic operation on two floatx values is equivalent (up to problems with double rounding) to the equivalent arithmetic operation on their backend representations, followed by a rounding of the result.

```
backend_float enforce_rounding(backend_float value) noexcept {
  const auto exp_bits = get_exp_bits(self());
  const auto sig_bits = get_sig_bits(self());

  if (exp_bits == float_traits<backend_float>::exp_bits &&
      sig_bits == float_traits<backend_float>::sig_bits)
    return value;

  bits_type bits = reinterpret_as_bits(value);
  auto sig       = (bits & backend_sig_mask) >> backend_sig_pos;
  auto raw_exp   = bits & backend_exp_mask;
  const auto sgn = bits & backend_sgn_mask;

  int exp        = (raw_exp >> backend_exp_pos) - backend_bias;
  const int emax = (1 << (exp_bits - 1)) - 1;
  const int emin = 1 - emax;

  if (!is_nan_or_inf(bits))
  {
    if (is_small(exp, emin))
      convert_subnormal_mantissa_and_exp(bits, sig_bits, emin, exp,
     sig, raw_exp);
    else
      sig  = round_nearest(sig, backend_sig_bits - sig_bits);
    if (significand_is_out_of_range(sig))
      fix_too_large_mantissa(sig_bits, exp, sig, raw_exp);
    if (exponent_is_out_of_range(exp, emax))
      bits = assemble_inf_number(sgn);
    else
      bits = assemble_regular_number(sgn, sig, raw_exp);
  }
  return reinterpret_bits_as<backend_float>(bits);
}
```

Listing 5.3: Main floatx rounding routine.

- Printing a floatx value to a stream is equivalent to printing its backend representation.
- Reading a floatx value from a stream is equivalent to reading it as a backend value and rounding the result.
- Converting a numeric type into a floatx type is equivalent to converting it into the backend type and rounding the result.
- Converting a floatx type into a numeric type is equivalent to converting its backend representation into that numeric type.
- Converting between two floatx types is equivalent to re-rounding the backend representation.

As a result, the non-trivial part of floatx remains hidden in the implementation, exposing an intuitive interface to the user which makes

floatx (semantics) behave as expected and supports a flat learning curve.

Creating floatx objects and casting between floatx objects (as well as between built-in and other floatx objects) should be as easy as creating and casting between built-in objects. To attain this, floatx defines a set of converting constructors for creating floatx objects and casting to floatx objects, as well as a set of conversion operators for casting from floatx objects to built-in types.

To support arithmetic and relational operations using the same syntax as that of built-in types, the library provides a complete set of arithmetic and relational operator overloads for floatx numbers. In addition, stream input and output operator overloads are also provided to simplify writing and reading floatx objects to and from C++ streams.

Supporting interoperability between distinct types (i.e., accommodating operations involving operands of different types) is more difficult. This requires extending the standard numeric promotion and conversion rules to include floatx numbers. The C++ definition of these rules relies on the concept of a common type, a rigorous definition of which, including the set of implicit conversions and promotions, can be found in the C++ standard [161].

In short, the common type for two floating-point types F and G is the more precise of the two, and the common type of an integral type I and a floating-point type F is the floating-point type F. All binary operations are performed by first converting both operands to their common type, and then performing the operation with converted values. The result of the operation is of the common type. In addition, if the integer being converted is out of the representation range of the common type (this can happen when converting a large integer to a low-precision floating-point type), the result of the operation is unspecified. Applying specifically these rules to floatx is impossible, since there are pairs of floatx types such that neither of them can be considered as more precise (e.g., *floatx<7, 9>* and *floatx<10, 6>*). Thus, floatx has to extend these rules in a way which does not modify their definition for standard types, while maintaining the desirable properties of those types for floatx's emulated types. Some of the properties that we consider crucial are the following:

- The common type of two operands of the same type T is the type T itself; that is, $common\_type(T, T) = T$.

- For any two floating-point types R and S, and objects of those types a and b, respectively, consider the statement $c = a + b$; (equivalently, any basic arithmetic operation). If c is of type T the final result stored in c is equivalent (up to effects of double rounding) to the infinitely-precise result of $a + b$, properly-rounded to type T.

- For any two floating-point types R and S, $common\_type(R, S)$ is the smallest type which has at least as many exponent and significand bits as both R and S. To remove ambiguities with the standard specification of a common type, floatx uses the following definition.

Definition 6 *(Common type for floatx). Let e_S and e_T denote the number of bits reserved for the biased representation of the exponent in the floating-point types S and T, and let m_S and m_T stand for the number of bits in the significand (mantissa) of the same types. For two floating-point types, S and T, floatx defines $common\_type(S, T)$ as a type with $\max e_S, e_T$ bits reserved for the exponent and $\max m_S, m_T$ bits for the significand. For a floating-point type S and an integral type I, $common\_type(S, I) = S$.*

Note that this definition satisfies all of the above mentioned desirable properties and keeps the original semantics of a common type unchanged for the built-in types. Since the floatx definition extends the original C++ definition, floatx only needs to implement the extensions for operations where at least one of the operands is of a type provided by floatx. This is done by providing more general operator overloads that can take any operands, as long as at least one of them is a floatx-provided type, and then convert them into their common type.

The common type is determined at compile-time, using a combination of meta-programming techniques including: trait classes for built-in and floatx-provided types, substitution failure is not an error (SFINAE) [161], and the std::enable_if standard C++ utility. For

```
template <typename backend_float = double, typename metadata_type =
    unsigned int>
class floatxr {
private:
  backed_float data;
  metadata_type exp_bits;
  metadata_type sig_bits;
};
```

Listing 5.4: Basic structure of `floatxr`.

implementation details, refer to the floatx source code[4]. The common type resulting from such an overload is always a floatx-provided type with the significand and exponent bits set to the values specified as in Definition 6. The backend type is set to the common type of backend types of the operands (if one of the operands is a built-in type T, its backend type is considered to also be T).

5.1.6 The floatxr class template

One downside of floatx is that the range and the precision of a type need to be known at compile time. Since the goal of floatx is to ease and accelerate the experimentation with low precision, this limit can sometimes be too restrictive. There are applications (e.g., Jacobi linear solvers [162]) for which it is interesting to start with low precision and increase the number of bits in small steps as the algorithm progresses. In this case, having a different type for each step will make the actual implementation very complex. Besides that, the intent to evaluate an algorithm with all significand sizes between S_1 and S_2, and return the optimal one is difficult to express, as it would either require the user to recompile the code for each size or use template meta-programming to write the compile-time equivalent of a loop. For this reason, we introduce another class template, floatxr, which allows the user to set the precision at runtime. Listing 5.4 displays the data layout of this type.

Specifying the precision at runtime comes at the cost of an increased memory footprint, and as such, floatxr does not fulfill the third

[4] Available at `https://github.com/oprecomp/FloatX` (Accessed, September 2019)

design goal specified at the beginning of this section. However, floatxr and floatx can be implemented to use the same code base, so there is only one rounding routine and one set of operators to maintain (and the compiler just optimizes the same generic function templates more efficiently when instantiating them for floatx than for floatxr). Also, floatxr types support the same operations and conversions as floatx types, and interoperate with floatx and built-in types. As stated in the following definition, the latter requires a minor revision of the common type, since the precision of floatxr is not known at compile-time.

Definition 7 *(Common type for floatxr). For two types S and T:*

- *If at least one of S and T is a floatxr type, their common type is a floatxr type whose backend type is the common type of S and T's backend types.*
- *Otherwise, the common type is deduced as specified in Definition 6.*

5.1.7 Notes on concurrency

Floatx variables maintain no state (i.e., there are no routines for setting or global variables holding the current working precision, since this is either a property of the types involved in an operation and/or a property of the data). The rounding operation itself also depends only on the backend and the destination types' properties. An important advantage of the stateless floatx compared with stateful floatxr, is that the framework can be safely used from many concurrent threads of execution, in a sense that no arithmetic or relational operation on floatx variables involves accessing any other non-constant data. The threads in question can either be operating system threads on a CPU or CUDA threads on a GPU. It should also be noted that the rounding operation and the functions employed in floatx consist of the same code in both the CPU and the GPU cases, up to certain integer compiler intrinsics and the necessary CUDA device function attributes. Since changing the precision of a floatxr variable at runtime requires modifying its two metadata components and re-rounding the value in the backend component, neither read nor write access to the variable from another thread should be allowed while the operation

is in progress. In general, there are no special guarantees of atomicity for any floatx type or operation. For example, an assignment of a floatx variable b to a involves: creating a temporary floatx with the backend value set to the one of b, re-rounding of that value in-place, and finally replacing a's backend component with the temporary's (up to any compiler optimizations that might be applied). The users should, therefore, rely on standard mutual exclusion primitives to avoid data races during the concurrent access to floatx variables of any type.

5.1.8 Advanced properties and performance of floatx

An additional advantage of the stateless floatx is that the memory size and the alignment requirements are both identical to those of the backend type. Thus, in certain read-only scenarios, an array of floatx variables can be directly passed, with just a pointer typecast, to a routine expecting an array of the backend type. A use case might be a pretty-printer routine, or a writer routine for a custom file format, possibly written in a different language. The easiest way to compute the mathematical functions for floatx is to obtain the result using the backend function, and round the result into a target floatx type. Of course, that could sporadically introduce results which are not correctly rounded, due to double rounding, where the routine itself is expected to be correctly rounded (e.g., sqrt). When representing a native floating-point type as a floatx type (in particular, floatx<11,52> represents double, and floatx<8,23> is equivalent to float), the final rounding operation is unnecessary (no-op) when that machine's type is equivalent to the backend one, or can otherwise be simplified by employing two native data type conversions. In both cases, the simplification can be triggered at compile time. The former case yields arithmetic performance close or equivalent (zero-overhead achieved by code inlining) to using the native backend type directly, including the possibility of automatic vectorization and other optimizations not generally applicable to floatx types. The latter case has not yet been implemented.

Overhead. It is evident from the control flow in Listing 5.3 that the complexity of rounding depends on the value being rounded as well as on the type constraints. Therefore, the performance of the

code is inherently dependent on the data itself. For example, the rounding operation from Listing 5.3, fully optimized and inlined after an addition a + b, with the variables declared as floatx<7,12> a and floatx<10,9> b, requires around 80 assembly instructions on an Intel Haswell architecture with the gcc 7.2.1 C++ compiler. However, not all of those instructions are executed in each rounding operation, due to possibly different code paths taken in each case. Furthermore, the alternative code paths make code vectorization almost impossible. Our next experiments provide an evaluation of the practical overhead introduced by floatx when using non-native data types, using the routines for the legacy[5] basic linear algebra subprograms (BLAS)-1 dot product (DOT) and BLAS-2 general matrix matrix multiplication (GEMM) converted to C++ and integrated with floatx. The testing machine for all performance experiments presented in this section (except when the target is a GPU) was an Intel Xeon E5-2630 v3 (Haswell) system, running at 2.40 GHz, with the gcc 7.3.0 C++ compiler and a 64-bit GNU/Linux.

Table 5.1 summarizes the performance of the DOT and GEMM operation for various data types. The two input vectors to the DOT kernel are of length $n = 10^9$ filled with pseudo-random elements in $[-1, 1]$. The GEMM kernel performs $\mathbf{C} = 2\mathbf{AB} - \mathbf{C}$ with input matrices of order $n = 3000$ and leads to a total of $2n^3 + O(n^2)$ floating-point operations. We observe that all floatx types exhibit a similar (but not the same) performance hit compared with the backend type, except when equivalent to it and the optimization described above is in effect. However, when the exponent range is the same as one of the backend types (11 bits for double), the rounding procedure from Listing 5.3 is faster than in the other cases. The measurements confirm that the complexity of the rounding operation varies with the data type of the parameters as well as with the data values themselves.

[5] Available at `http://www.netlib.org/blas` (Accessed September 2019)

Table 5.1: Runtime performance in MFLOP/s for DOT BLAS-1 and GEMM BLAS-3 operations with various floatx and native types.

Type	$T_{5,10}$	$T_{10,13}$	$T_{11,44}$	$T_{8,23}$	float	$T_{11,52}$	double
DOT	128.8	127.7	149.4	132.6	1999.1	1714.8	1715.2
GEMM	125.1	125.7	149.8	128.4	5751.0	2316.8	2652.0

5.2 Numerical analysis of applications

5.2.1 PageRank

The memory-bound PageRank [2] iteratively computes the node score given the topology of a directed graph as sparse adjacency matrix of size $n \times n$ with z non-zeros as detailed in Section 2.1.1. Due to the iterative nature of the algorithm and its known numerical stability [82] it becomes the preferred candidate for computations based on reduced precision floating-point formats. Sparsity-aware implementations ($n < z < n^2$) reach a time complexity of $O(Iz)$ and the memory complexity is $O(z)$, where I denotes the input dependent number of iterations till convergence. PageRank is memory-bound since each iteration accesses z matrix entries while performing constant work $O(1)$ per entry.

In this section, we demonstrate two features of transprecision computing: First, we empirically demonstrate that the numerical behavior of the algorithm converges with the same number of iterations and reduces the residual to the same quality level as the reference implementation. Second, we estimate the potential gains by using reduced precision over the baseline.

We aim to use a fixed precision T_{w_i,t_i} for all operations (loading, storing, and arithmetic computations) in the i-th iteration to reduce the bottleneck bandwidth. Memory access improves by bit-width reductions of the loaded data. Such an approach leads to a faster execution of a single iteration if the underlying hardware supports transprecision. However, due to computing with reduced precision, intermediate results change due to quantization effects. It is non-trivial how errors propagate through the computations. Additionally,

PageRank's iteration count is controlled with a data-dependent residual. If quantization effects cause the algorithm to stay longer inside the iteration loop, potential execution performance gains per iteration might get quickly lost.

An excursion on forward error bounds

Even though we can analytically compute upper error bounds caused by quantization and propagation effects, we show how they quickly diverge such that they do not provide results of practical interest. For example, for a matrix-vector multiplication of matrix size $n \times n$ and vector dimension n, the output of each values is produced by a sum of the dot product of length n. Since the matrix is sparse, only $n_{row} \in [0, n]$ non-zero entries contribute to the summation. We denote with $\hat{x}$ and $\hat{a}$ the exact, scalar values of the iteration vector and the system matrix. We refer to the normal, and henceforth, quantized values, with x and a respectively. We define δ_x and δ_a as the absolute quantization error of one scalar, i.e., $x = \hat{x} + \delta_x$ and $a = \hat{a} + \delta_a$. Δ_x and Δ_a refer to the maximal, per-scalar, quantization error. For each scalar multiplication we have the following error propagation,

$$x \cdot a = (\hat{x} + \delta_x) \cdot (\hat{a} + \delta_a) = \hat{x} \cdot \hat{a} + \hat{x}\delta_a + \delta_x\hat{a} + \delta_x\delta_a. \tag{5.1}$$

Using the per-scalar quantization error bounds $\delta_x \leq \Delta_x$, $\delta_a \leq \Delta_a$, and without loss of generality, we know that in PageRank the iteration vectors and the system matrix is normalized such that we can ensure that $x \leq 1$ and $a \leq 1$ holds, we obtain the following per-product error bound:

$$|x \cdot y - \hat{x} \cdot \hat{y}| \leq \Delta_x + \Delta_a + \Delta_x\Delta_a. \tag{5.2}$$

Equation (5.2) holds per each scalar multiplication. Even without assuming additional errors due to the summation, adds up linearly in the length of the dot product n_{row}. The iterative nature of PageRank causes the errors to propagate such that we upper bound errors per-value with the following recursion,

$$\Delta_x[i+1] \leq n_{row} \left(\Delta_x[i] + \Delta_a + \Delta_x[i]\Delta_a\right), \tag{5.3}$$

where $\Delta_x[i]$ denotes the iteration error bound for the i-th iteration. Note, since the system matrix $\mathbf{A}$ is constant, the error bound Δ_a does not grow. Explicitly, the given upper error bound grows at least exponentially in the number of iterations,

$$\Delta_x[i] \quad \geq \quad (n_{row})^i \, \Delta_x[0], \tag{5.4}$$

if we assume $\Delta_a = 0$ to simply resolve the recursion. Even if we just assume a toy example with at most $n_{row} = 10$ entries per row and we only iterate for 10 iterations, and we operate with floating-point 32-bit precision, which results in $\Delta_x[0] := 2^{-24}$, we would achieve an upper bound of $\Delta_x^{10} = 10^{10} \cdot 2^{-24} = 596.05$. Since we expect the final result to be normalized and we need the values to be enough distinguishable to not change the sorted ranking, upper error bounds require to be at least four to five orders of magnitude smaller to be of practical interest. This consideration highlights the importance of the methodology, to use floatx instead, to study the numerical behavior of reduced precision on PageRank.

Transprecision design for PageRank

PageRank is numerical stable [82]. That causes, even in the presence of numerical perturbations, the algorithm to converge in all practical test-cases. Those conditions allow that the algorithm works with any precision level and converges to machine precision imposed by the numerical representation.

To that end, we adaptively increase the precision during subsequent iterations. Since precision changes cost n casts for the iteration vector and z casts for the adjacency matrix, we aim to lower the number of precision adaptions. Henceforth, we fix the number of precision changes such that $T_{i+1} = T_i$ holds for most consecutive iterations, where T_i denotes the data type used in the i-th iteration. If we choose a constant set of working precisions with a limited cardinality, the total casting overhead amounts to a complexity of $O(z)$ which is small compared to the full algorithm complexity $O(I * z)$.

The critical question that is required to be studied, is how to adapt the precision and if overall performance gains are achievable. In other words, the number of iterations achieved in practice with reduced precision computing requires to compete with the baseline.

Table 5.2: Impact of $T_{w,t}$ on iteration count for PageRank by using the GPU supported data types.

Data	Ref	#Itr.	$T_{5,10/8,23/11,52}$	Cost	Saving
Synth_1	24.3	31.1	3.0–17.1–11.0	20.3	16.4%
Synth_2	148.6	150.4	10.7–66.8–72.8	108.9	26.7%
Real	251.8	251.6	19.3–77.8–154.5	198.2	21.3%

Table 5.3: Impact of $T_{w,t}$ on iteration count for PageRank by using 8 data types that are aligned to 8-bit each.

Data	#Itr.	$T_{4,3/5,10/6,17/7,24/8,31/9,38/10,45/11,52}$	Cost	Saving
Synth_1	51.5	1.0–7.1–6.4–12.9–9.9–11.2–2.0–1.0	28.1	-15.7%
Synth_2	157.8	1.0–12.6–36.5–33.2–29.7–31.5–12.3–1.0	87.5	36.8%
Real	252.8	1.0–19.9–50.5–28.7–31.7–31.9–20.3–68.7	168.6	33.0%

The baseline PageRank stops whenever the relative residual error $\xi_i = \|\mathbf{p}_{i-1} - \mathbf{p}_i\|_1$ is reduced below a given threshold ϵ. To automatically switch precision, we monitor the convergence rate ξ_{i-1}/ξ_i and we increase the working precision whenever the convergence rate ξ_{i-1}/ξ_i is slower than a given threshold ρ. The iteration vector $\mathbf{p}$ is re-normalized after precision changes to recover distortions of the invariant caused by the previous (inexact) representation. Our changes allow replacing major parts of computation into lower precision while still achieving the same (strict) precision requirement ϵ as the baseline.

Table 5.4: Measured GPU kernel performance for dense PageRank iterations using floating-point types *half*, *float*, and *double*.

Opt. Speedup	*half*: 4×	*float*: 2×	*double*: 1×
Measured Speedup	**3.44×**	**1.91×**	**1×** (Reference)

Transprecision results for PageRank

Table 5.2 shows average iteration counts for the baseline and GPU supported types, whereas Table 5.3 states results when running with eight formats. Linear interpolation between *half* and *double* determines the configuration of w and t used in the 8-bit aligned intermediate formats. We run on two synthetic datasets (the existence of link pairs is i.i.d. Bernoulli $p_1 = 0.01$ and $p_2 = 0.001$ distributed) and real graphs extracted from the web [163]. Even in the baseline, input dependent convergence causes different average iteration counts. Extra iterations caused by precision control adaptation and quantization caused perturbations in the data, explain slightly increased iteration counts for the transprecision versions. Costs and improvements are estimated based on linear bit-width reduction gains per iteration. For GPU type restricted execution gains of 20% are realistic, where higher gains are achieved for slower converging instances. Using a finer granularity for intermediate formats helps to improve gains (from 21.3% to 33% for real data) where major workloads in formats between *float* and *double* occur. Table 5.4 presents measured GPU per iteration timings obtained for implementations based on *half* types and compute unified device architecture (CUDA) provided intrinsic functions supporting that type. Measurements come close to postulated optimal gain factors.

5.2.2 BLSTM

BLSTM is an instance of a deep learning model trained to perform the task of optical character recognition as explained in Section 2.1.2. With this example we demonstrate how transprecision computing applies at three conceptually different levels: First, the model parameters can be compressed to reduce the overall memory footprint, second, the model input can be provided with reduced precision, and third, the actual arithmetic computations are performed with reduced precisions.

Transprecision design for BLSTM

Matrix-vector operations dominate the BLSTM [3]. The BLSTM updates two LSTM cells in a forward and backward pass, merges

results with a dense layer and finally predicts with a softmax output layer. Even though LSTMs are evaluated in a recurrent fashion, they are fed with new inputs at each iteration and we consider them as fully loop unfolded, plain feed-forward based computations. In contrast to PageRank, artifacts caused by numerical errors due to quantization might potentially propagate through the output and cause wrong results. However, the known error resilience of deep learning methods [3, 164] allows recovering from imprecision caused by numerical representations. Addressing the granularity of the number format exploration, we decided to study the following aspects of inputs, weights, and computation separately. For the arithmetic computations, we considered the following assumptions: First, all weights belonging to a chunk of data should have the same data type and second, the output data type of a module has to match the input data type of the next module. Since the LSTM cell passes values recursively through the same module, the stated assumptions resolve in basically using one precision for dealing with input/outputs of LSTM cells. Since BLSTM consists of two independent LSTM cells responsible for the forward and backward computations only, we decided to use data type precision levels homogenous among all LSTM cells. The dense layer that merges LSTM outputs used different internal precisions for computations that are performed independent of the operation of the LSTM cells. Since parameters for the merging dense layer consist of less than 5% of model parameters, we decided to assign the same precision as used for the majority of all parameters stemming from LSTM cells.

Transprecision results for BLSTM

The first experiment executes all parts, including the model parameters, the inputs, and all intermediate computations, with a global reduced precision data type $T_{w,t}$. Figure 5.3 shows the final achieved accuracy when performing a full grid search among all global configurations $T_{w,t}$ by using a variable bit-width to encode exponents with w and the mantissa with t bits. The full grid search reveals a sharp transition between good and bad operation. Since the original BLSTM is designed to run with IEEE 754 32-bit float values, increasing range or precision above $T_{8,23}$ does not alter the result and explains the plateau

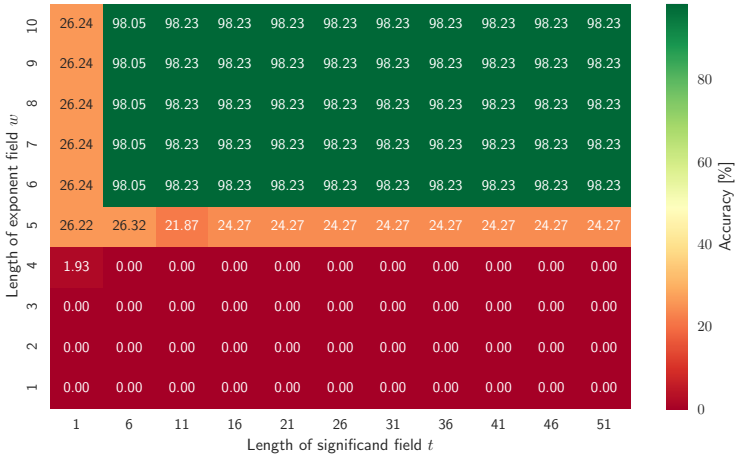


Figure 5.3: Top1 accuracy of BLSTM when operating with type $T_{w,t}$. The baseline runs with $T_{8,23}$ which explains part of the plateau at around 98% accuracy. BLSTM operates close-to-perfect with further reduced formats.

between $T_{8,23}$ and $T_{11,52}$. Interestingly, configurations with less than 32-bit wide data types do almost not exhibit a smooth transition in performance, however, a sharp transition to a full-fail of the algorithm is present. We identify $T_{6,6}$ as an optimal global configuration that operates well. $T_{6,6}$ is considerably narrower than $T_{8,23}$ (*float*) used to execute the baseline.

Table 5.5 presents accuracies obtained for selected operation points. Operating with a 16-bit format coded as $T_{8,7}$ [165] outperforms the $T_{5,10}$ (*half*) format. Individually reducing input encoding for weights (W) and image (I) down to $T_{3,1}$ has a marginal effect on the result. Since simultaneously using $T_{3,1}$ for weights and images reduces accuracy, we used $T_{4,1}$ that enables decent accuracy. Profiling shows that multiply-accumulate (MAC) operations from dot products contribute more than 80% of all executed operations. We suggest to use $T_{5,1}$ arithmetic for multiplication and a $T_{5,10}$ arithmetic for accumulation. The last configuration shows that replacing the remaining parts with

narrow formats allows us to compute with an average bit-width of 12.2 bit and still getting an accuracy of about 98%.

5.2.3 GLQ

The GLQ numerically computes the integral over a function. Due to the inherent approximative nature, even when running with full precision, the quality of the integration routine is improved when splitting the integration domain into more intervals or when employing a higher-order polygon during the GLQ routine.

Transprecision design for GLQ

The GLQ kernel offers different options to introduce reduced precision data types. Since our evaluation is based on the Genz functions [103] for which we can compute the analytical solution to evaluate numerical quality, we do not want to over-tune the solutions for those

Table 5.5: Impact of reduced precision on BLSTM Top1 accuracy: Reference results, global scans, input quantization effects and proposed configurations.

Setting	Weights	Img.[1]	Operations	Accuracy
Ref [3]	$T_{8,23}$	$T_{8,23}$	100% in $T_{8,23}$	98.2337%
Ref [3], Fig. 6	16-bit fixed		See [3] IV-C	97.9794%
Ref [3], Fig. 6	5-bit fixed		See [3] IV-C	97.5821%
$T_{5,10}$ (*half*)	$T_{5,10}$	$T_{5,10}$	100% in $T_{5,10}$	21.4392%
$T_{6,6}$, see Figure 5.3	$T_{6,6}$	$T_{6,6}$	100% in $T_{6,6}$	98.0536%
$T_{8,7}$, see [165]	$T_{8,7}$	$T_{8,7}$	100% in $T_{8,7}$	98.1890%
Quantized W	$T_{3,1}$	$T_{8,23}$	100% in $T_{8,23}$	98.0692%
Quantized I	$T_{8,23}$	$T_{3,1}$	100% in $T_{8,23}$	98.1181%
Quantized W&I	$T_{3,1}$	$T_{3,1}$	100% in $T_{8,23}$	96.7730%
Quantized W&I	$T_{4,1}$	$T_{4,1}$	100% in $T_{8,23}$	98.1660%
Modified MAC (Average Width: 15.7-bit)	$T_{4,1}$	$T_{4,1}$	40.7% in $T_{5,2}$ 40.7% in $T_{5,10}$ 18.6% in $T_{8,23}$	98.1905%
Proposed (Average Width: 12.2-bit)	$T_{4,1}$	$T_{4,1}$	40.7% in $T_{5,2}$ 40.7% in $T_{5,10}$ 18.6% in $T_{6,6}$	97.9969%

[1] data type applied to the input images

Table 5.6: Errors of GLQ for IEEE 754 standard formats and the 16Alt format.

Genz function	*double* $T_{11,52}$	*float* $T_{8,23}$	*half* $T_{5,10}$	*16Alt* $T_{8,7}$
1	7.0e-16	2.6e-07	1.0e-02	1.5e-01
2	3.3e-16	1.5e-07	1.5e-02	1.3e-01
3	4.3e-16	2.6e-07	1.5e-02	1.8e-01
4	4.6e-16	4.1e-07	2.0e-02	1.1e-01
5	5.2e-14	1.6e-07	2.1e-02	2.5e-01
6	4.2e-07	4.8e-07	8.1e-03	1.0e-01

specific functions. That is the reason we have decided to use one single data type throughout, including all coefficients for the GLQ, all input function parameters, the function input argument, the output of each function evaluation, all internal variables used to perform the summation and splitting of interval borders, and the final result. Since GLQ computes a numerical integration and we know the analytical solutions in the test setup, we characterize the normal numerical behavior of the GLQ kernel based on primary parameters, such as the order of the GLQ kernel and the number of subintervals the original domain is split into. We argue that if the transprecision can deliver results of similar quality as in the case of floating-point that has an error inherently present due to the approximate nature of the integration, we can say that both solutions achieve similar quality. In other words, the reduced precision result is required to be close to the—analytically computed—exact value and not the full precision computed GLQ approximated value.

Transprecision results for GLQ

In contrast to PageRank and BLSTM, we can test the quality of the GLQ routine with test functions that are constructed such that the solution is given by an analytical expression. Since we know the exact solution, we do not compare reduced precision data types against the full precision code variant, but we rather compare all obtained results of the GLQ routine against the exact solution. Two parameters of

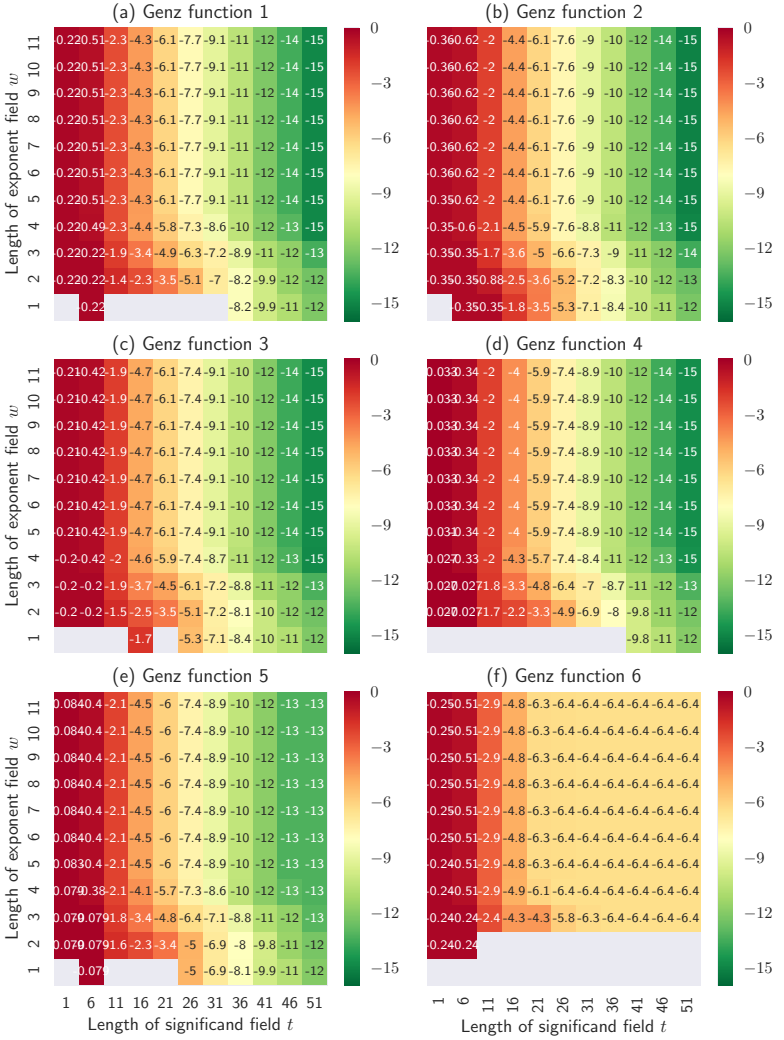


Figure 5.4: Global effect on GLQ when using reduced precision of data type $T_{w,t}$. Shown are the mean of the absolute error obtained for ten runs fore each of the six Genz functions. Gray colors indicate infinity or NaN results in at least one test case. GLQ does not rely on large dynamic ranges, results stabilize when using four or more exponent bits.

the GLQ kernel affect the performance and quality trade-off in our setting, the number of intervals N_I the integration domain is split into and the number of support point used in the core GLQ routine N_O. Since the GLQ kernel is implemented with two nested control loops that perform work proportional to the defined parameters, the total time complexity is of order $O(N_I N_O)$.

We evaluate the numerical effects of transprecision computing for the conservative configuration of control parameters $N_I = N_O = 1024$. Table 5.6 states the obtained errors for the IEEE 754 formats and the proposed 16Alt format [154]. Errors are measured against the analytical solution as absolute error expressed as base-ten logarithm over the average obtained when running with ten uniform sampled function parameter instances. The discontinuity in Genz Function 6 causes to challenge the GLQ routine and final precision is limited by how the intervals are split relative to the position of the discontinuity, which explains the worse result when compared to the other Genz test functions for the *double* precision case. However, computing with *float* 32-bit the machine precision is of about 6 decimal digits and achieved in all test cases. Note, that the challenging Genz function 6 performs similar to the other functions. In the GLQ kernel, the IEEE 754 half is outperforming the 16Alt format because the internal routines do not require an extended dynamic range, henceforth the reduced number of mantissa bits of 16Alt compared against half cause to harm the results.

Figure 5.4 shows the global behavior of the result when employing all computations with a specific format. As observed in Table 5.6, the discontinuity of Genz test function 6 is limiting the final accuracy independent of the precise choice of the number format. Grey colored fields indicate that at least one of the produced computation either triggered an overflow to infinity or a NaN. Very small formats tend to behave that way, independent of the function used to test the behavior of the GLQ routine. Again, Genz test function 6 causes the strictest test case. As postulated, increasing the number of exponent bits increases the dynamic range and covers more case of the computations to be in the valid range. In the GLQ kernel, increasing the exponent beyond four bits does increase the precision of the result. Increasing the number of mantissa bits increases the obtained accuracy and that holds over the full range of exponent encodings. Note, as corner case,

even when the exponent is one single bit increasing the mantissa enlarges the dynamic range due to subnormal encoded represented numbers, which recovers from infinities and NaNs in some cases.

5.3 Summary and conclusion

In the first part of this chapter, we design and implement floatx to emulate reduced precision floating-point arithmetic and data types. Floatx features the following:

- **Programming interface.** Floatx is easy-to-use and minimally-intrusive to facilitate an incremental transformation of numerical applications.
- **Performance.** Floatx relies on hardware-supported floating-point types in the backend to preserve efficiency. Furthermore, it incurs no storage overhead by maintaining the size of the emulated datatype shorter than (or equal to) that of the backend datatype.
- **Expected semantics**. The arithmetic in floatx adheres to the round-to-nearest, ties-to-even rule in the standard IEEE 754 whenever feasible, and the interoperability of variables follows the data type casting convention in C++.

In the second part of the chapter, we apply numerical evaluations with floatx to showcase transprecision computing. We achieved the following results:

- **PageRank.** The iterative nature of the kernel allows to adaptively change the precision as the algorithm converges towards the result. We demonstrate that the residual can be reduced to the same quality level without increasing the total number of iterations. That way, with linear bit-width models, we expect above 30% performance gain achieved with transprecision computing without quality loss in the final result.
- **BLSTM.** Even though neural network based approaches potentially propagate errors towards the output and might degenerate

the accuracy of the model, their inherent error resilience allows for coarse-grained quantization with only minor degenerations in the final quality. In the case of BLSTM, computing with an average of 12.2-bit achieves similar to 32-bit reference precision. Different causes of quantization error can be tuned independently, we demonstrate how we can compress the input weights down to 6-bits achieving similar as the reference accuracy.

- **GLQ.** The numerical nature of the GLQ kernel causes different approximation errors even in the reference when computing with full *double* 64-bit precision depending on the parametric settings. Our transprecision analysis demonstrates a smooth error behavior. Since most intermediate results are normalized, as few as 4 exponent bits are enough to compute results of reference quality.

In this chapter, we developed the floatx library to perform numerical studies on applications. The improvements on three kernels empirical demonstrate the success of transprecision computing by achieving accurate results with reduced bit-widths. The floatx library is self-contained and decouples complexity away from the applications under considerations. The modular approach allows reusing floatx and enables numerical studies in any application. We judge our findings as a strong indication that similar improvements are achievable for iterative solvers, stencil-based computations, weather, physic, or mechanical simulations among many more. A detailed numerical study opens the opportunity to identify too conservative segments of algorithms or applications. Those insights help to map current applications on systems that are supporting multiple precision levels with controlled quality. Additionally, fine-grained emulations allow to further improve the next generation of computing systems.

In the next chapter, we extend the results for the domain of deep learning to focus on scalability, easy-to-use, and generalization of transprecision computing.

Chapter 6

Floatx for deep learning

In Chapter 6 we presented floatx, a C++ header-only library that allows elaborating reduced precision formats. In this section, we explain concepts and technical details on how we integrate floatx into PyTorch, a common deep learning framework. The main purpose of the integration of floatx into PyTorch [62] is to enable numerical experimentation. Flexibility, re-usability, and modularity of code are import design considerations since deep learning is a fast-moving field such that similar experiments and considerations should be able to be repeated on a novel and by third-party defined networks. At that point, we anticipate that the natural error resilience of neural networks allows the trimming of numerical representations. We are interested in understanding numerical behavior and the mapping between used number formats and achieved accuracy of specific models. This section provides modular building blocks that are further used in the next chapter to draw conclusions on how trade-offs achieved with transprecision compare against strong baseline methods that are achieved by solving the NAS problem for specific constraints. Integrating floatx into deep learning enables experimentations to demonstrate and to seek opportunities including model weight compression, activation compression to reduced transfer times, and to experiment with the numerical behavior of specialized computing devices featuring reduced precision.

6.1 Integrating TP into deep learning

Before discussing the internals of the integration of floatx, we define what aspects matter to apply transprecision computing. Table 6.1 defines the key aspects with options that conceptionally characterize how a reduced precision solution is operating. We define three use cases of transprecision computing.

- First, the case *TP at inference* introduces transprecision at inference for a trained model to achieve better execution performance.
- Second, the case *Training accounting for TP at inference* modifies the training routine to account for precision changes and to optimize the model accounting on how transprecision computing affects intermediate results.
- Third, the case *TP at training* performs the full training with reduced precision in order to accelerate the training routine itself.

Transprecision computing occurs in two modes, extrinsic and intrinsic. Similar to reference work [166], we define extrinsic quantization as applying it to full tensors at input and output level of operations. Namely, this includes weight compression and activation map compression. Intrinsic quantization refers to the situation where it is applied after each scalar operation of a specific kernel implementation. Extrinsic quantization is simple, modular, and efficiently exploitable by implementing the quantization operator and reusing the existing framework operations. An intrinsic implementation, however, requires to reimplement the low-level kernels to have full control over all arithmetic precision aspects. Finally, the available hardware at runtime determines if the considered transprecision computations are emulated or natively run. In this section, we focus on integrating floatx as emulation into code that runs on traditional hardware without reduced precision support.

We define a model as directed acyclic graph (DAG) as $M = (V, E)$ where V is the set of nodes defining the kernels and E is the set of edges that defines the flow of tensor data between kernels. Each

Table 6.1: Concepts to integrate TP into deep learning

Aspect	Variants	Description
case	*TP at inference*	TP features are used when the model is deployed during inference. How the original model is trained is not irrelevant, it might be provided by a third-party, or trained with regular arithmetic.
	Training accounting for TP	TP features are integrated to affect the forward computations of the compute graph. However, the training routine accounts for the reduced precision. Requires a specialized training routine.
	TP at training	TP features are integrated the forward and backward nodes of the compute graph to accelerate the training.
mode	*extrinsic*	Quantization at tensor level of input and output of kernels. Reuses natively implemented kernels at full precision.
	intrinsic	Emulates quantization at all arithmetic levels. Requires to reimplement kernels to have full precision control of internal arithmetic operations.
runtime	*emulated*	TP behaviour is not supported on current HW. The emulation evaluates numerical aspects of the reduced precision proposal.
	real HW	HW supports TP features and achieves measurable execution performance gains.

vertex $v \in V$ defines a kernel operation that is characterized by a list of inputs, internal trainable weights, and an output. Typical kernels are element-wise activation functions, elementary operations (such as addition, subtraction, element-wise multiplication, ...), dense layers (resulting in matrix multiply between input and weights), and convolutional layers, reshaping, shuffling, indexing, cropping, merging, and stacking operations among more specialized kernels used in recent models. Next, we define the quantization operator $Q_{w,t}(.)$ as element-wise casting values to a reduced precision type $T_{w,t}$ with w exponent bits and t mantissa bits. The quantization operator

Table 6.2: Introducing TP into the inference compute graph.

Weight	TP-mode	Impl.[1]	Description
Weight compression	*extrinsic*	$Q_{s,t}$	Compresses the memory consumptions of trainable parameters.
Activation compression	*extrinsic*	$Q_{s,t}$	Compresses the dataflow between kernels. Allows to reduce local transfer times, improves cache efficiency, shortens communication times in distributed systems.
TP arithmetic	*intrinsic*, *extrinsic*	$Q_{s,t}$, specific	All benefits as above and gains due customized low precision arithmetic.

[1]Implementation requirements. $Q_{s,t}$ states that the parameterized quantization serves that case.

is applied to data in the natively stored format and the output is returned in the native format, namely in the floating-point 32-bit standard format. Table 6.2 shows the available options to modify the compute graph of a model in order to achieve transprecision variants thereof. Transprecision applies at three levels, by compressing weights, by compressing activations, and by changing the arithmetic in computing units. The extrinsic approach, with the implementation of $Q_{w,t}(.)$, covers the first two use cases. However, the third use case requires a closer look at the requirements of the numerical evaluation framework. To keep the design simple and clean, extrinsic evaluations are modular and lead to efficiently running emulations. However, they might not fully adequately emulate the final bit-true behavior of the target system. The only way to enforce deterministic and bit-true emulations is to rewrite the kernels with customized code. Since that involves to write, debug and maintain a large amount of code and adds computational overheads, we favor emulations that are performed with the extrinsic approach. In the following, we study conditions that guarantee that the intrinsic and extrinsic approach deliver the bit-true equivalent result.

6.1.1 Arithmetic free and elementary kernels

We define the *operation internal DAG* as input to output scalar level dependency graph of a specific implementation of a kernel. We refer to the internal chain length as the number of arithmetic operations that occur on the longest input-to-output path. We claim equivalence of the intrinsic and extrinsic approach if, and only if, the internal chain length is zero or one. Into the class of zero internal chain length operations fall kernels that do not perform arithmetic operations on the data, typical instances are reshaping, concatenation and shuffling operations. Unity chain length kernels do not rely on internal intermediate results, henceforth the extrinsic quantization provides enough control over precision, typical instances are all element-wise operations. In contrast, the key kernel operations such as convolution and dense layers have an internal chain length that is larger than one, leading to intermediate results where precision can not be controlled with an extrinsic approach. On those operations, sequence ordering and double-rounding effects might cause different results between intrinsic and extrinsic approaches.

6.1.2 High precision accumulator assumption

To study and discuss the difference between the *intrinsic* and *extrinsic* approach we perform the full analysis on the dot product kernel. We define the dot-product of two one-dimensional tensors $\mathbf{w}$ and $\mathbf{x}$ of length n as follows:

$$\mathrm{DOT}(\mathbf{w}, \mathbf{x}) = \sum_{i=0}^{n} \mathbf{w}_i \cdot \mathbf{x}_i. \tag{6.1}$$

Using the extrinsic approach with customized precision data types for weights, input, and output (named as T_W, T_{IN}, and T_{OUT}, respectively) leads to the following:

$$Q_O(\mathrm{DOT}(Q_W(\mathbf{w}), Q_I(\mathbf{x}))) = Q_O\left(\sum_{i=0}^{n} Q_W(\mathbf{w}_i) \cdot Q_I(\mathbf{x}_i)\right). \tag{6.2}$$

In (6.2) the summation is stemming from the existing kernel and is performed in the native full precision format. In contrast, using the

intrinsic approach full control of all computing aspects are left to the developer. Equations (6.3), (6.4), and (6.5) define a fully specified transprecision implementation with an intermediate precision of T_U used as multiplier output and an accumulation precision of T_V used in the summation:

$$\forall i \in [1, n]: \qquad t_i = Q_U\left(Q_W(\mathbf{w}_i) \cdot Q_{IN}(\mathbf{x}_i)\right), \tag{6.3}$$

$$\text{SUM\_TP}(t_1, ..., t_n) = Q_V\left(...\left(Q_V(t_1 + t_2) + t_3\right) + ... + t_n\right), \tag{6.4}$$

$$\text{DOT\_TP}(\mathbf{w}, \mathbf{x}) = Q_O\left(\text{SUM\_TP}(t_1, t_2, t_3, ..., t_n)\right). \tag{6.5}$$

Equations (6.3) to (6.5) are equivalent to the extrinsic approach, if the intermediate results T_U and T_V are at full precision $T_{8,23}$, and the summation sequence in Equation (6.2) matches the ordering in Equation (6.4). To guarantee deterministic and bit-true equivalence among the emulation and the potential real operation on the target device, low-level implementation details such as the summation order has to be fixed since floating-point arithmetic does not obey associativity laws due to double rounding effects. Note, that even when requiring intermediate values at full precision, all multipliers can be optimized since they operate with reduced precision inputs. In some implementations, it is affordable to assume the presence of a full precision accumulation unit. With full precision accumulation and the same sequence order as in the baseline, the intrinsic emulation approach delivers the equivalent results as the extrinsic implementation. However, we favour the extrinsic over the intrinsic evaluation due to its implementation simplicity and efficiency of evaluation.

Even though the dot-product kernel is rarely used on its own, we argue that the same fundamental considerations generalize to more kernels, especially including dense layers and convolutional layers. First, we know that the matrix-vector or the matrix-matrix operation can be written as independent scalar products of input rows and columns. This argumentation allows to directly apply the numerical discussions from above. The traditional dense (or sometimes called

linear) layer is implemented as $\mathbf{y} := \mathbf{x}\mathbf{W}^T + \mathbf{b}$ which has an optional additive bias term. The above discussion applies by extending the summation with the bias term.

Similar considerations apply to convolutional layers. In a direct implementation, each scalar value of the output map is obtained as a summation over the product of filter coefficients multiplied by the actual input values. Moving the filtering masks to different positions results in independent computations that follow the same pattern. Note, that typical parameters of a 2D convolutional filter, such as applying different strides, padding, dilation, or even grouping factors affect how input-pixels are selected but the underling computations remain as the sum-of-products pattern.

The high precision accumulator assumption adequately covers the case where vectorized instructions extend an instruction set of a general-purpose processing unit. For example, 4- or 2-way reduced precision vectorized inputs fit into a single 32-bit data word. Supporting an additional scalar full-precision accumulation is enough to efficiently implement dense and convolutional layers.

6.1.3 The intrinsic versus extrinsic approach

Even though the intrinsic and the extrinsic approaches are fundamentally different, we legitimate that the extrinsic behaviour is enough to understand the fundamental numerics of deep learning models. To that end, we provide an example to understand theoretical error bounds and the statistical influence of quantization errors.

$$\begin{aligned} R &= x_1 + x_2 + x_3 + ... + x_n \\ R_{extrinsic} &= Q_I(x_1) + Q_I(x_2) + ... + Q_I(x_n) \\ R_{intrinsic} &= Q_A\left(...Q_A\left(Q_A(Q_I(x_1)) + Q_I(x_2)\right) + ... + Q_I(x_n)\right) \\ r &= Q_O(R) \\ r_{extrinsic} &= Q_O(R_{extrinsic}) \\ r_{intrinsic} &= Q_O(R_{intrinsic}) \end{aligned} \tag{6.6}$$

In the provided example we compute the sum R over n different values x_i for $i \in [0, n]$. We assume three levels of precision Q_I, Q_O,

and Q_A that define the data types used for the input, output, and the accumulation respectively. R, $R_{extrinsic}$, and $R_{intrinsic}$ denote the exact result, the extrinsic, and the intrinsic intermediate result in full and accumulation precision. The final results r, $r_{extrinsic}$, and $r_{intrinsic}$ are obtained after quantization of the intermediate results into the final output representation. We define the quantization error of a value x as $\delta := x - Q(x)$. We obtain the maximum quantization type-dependent error $\Delta := \max_{x \in \mathbb{R}} |x - Q(x)|$ by maximizing δ over the worst occurring combination. Each quantization error has an upper bound $|\delta| \leq \Delta$. Δ_I, Δ_O, and Δ_A state the input, output, and accumulation quantization error bounds. If we assume that all double-rounding effects occur to the worst possible extent, then we obtain trivial upper bounds on the errors:

$$\begin{aligned} |R - r| &< \Delta_O, \\ |R - r_{extrinsic}| &< n\Delta_I + \Delta_O, \\ |R - r_{intrinsic}| &< n(\Delta_I + \Delta_A) + \Delta_O. \end{aligned} \tag{6.7}$$

The error bounds in (6.7) demonstrate that the effects of the output representation are one complexity class lower than the effects stemming from the input or accumulation precision. Since at some point layers are evaluated in sequence, we assume to use the same data types for input and output, assuming that $\Delta_I = \Delta_O$ holds. We conclude that the extrinsic approach is dominated by the input quantization level and in the order of $O(n\Delta_I)$ and the extrinsic approach is in the order of $O(n(\Delta_I + \Delta_A))$. Assuming full precision accumulation as in Section 6.1.2 we achieve $\Delta_A = 0$ or at least $\Delta_A << \Delta_I$ and the intrinsic and extrinsic error bounds coincide.

The error formulation of the intrinsic approach demonstrates that adding guard bits to increase the precision of the accumulation helps to reduce error to a level where it is solely determined by the input error that is already reflected with the extrinsic approach. Since adding one additional bit to the mantissa field halves the maximal quantization error, the total error is dominated by $O(n\Delta_I(1+1/(2^g)))$ where we express the accumulation quantization error relative to the input quantization error $\Delta_A := \Delta_I/(2^g)$ where g denotes the additional number of guard bits used additional to the input data type.

More important, even if we do not use any additional guard bits ($\Delta_I = \Delta_A$) we can upper bound the error of the intrinsic approach to be as twice as large as the error of the extrinsic approach:

$$\Delta_I < \Delta_I \left(1 + \frac{1}{2^g}\right) \leq \hat{\Delta}_I := 2\Delta_I. \tag{6.8}$$

We realize that twice the original error is equivalent to performing the original extrinsic numerical study using one bit less in the mantissa bit field. We conclude that if we understand the numerical behavior for a full grid search of input quantization levels that scale exponentially in the number of mantissa bits $\Delta_I \propto 2^t$ we encapsulate the error bounds of the intrinsic case between to anchor results obtained at the finest resolution of the extrinsic analysis. This statement motivates us and other researches to perform the error analysis in a modular, simple, and compute efficient way.

We are fully aware of the limitations of the above considerations. First, the error bounds grow linear in the chain length causing an error bound growth belong all limits of practical interest, especially if we consider tiny formats where the initial quantization error is already large. In contrast, the very same argument favors extensive empirical numeric experiments to understand the effects of introduced quantization at the application level. Second, even if we understand the growth of the error bound real achieved quantization errors are more optimistic since they might cancel each other.

To complete the comparison between intrinsic and extrinsic quantization, we add a synthetic generated experiment that validates our thinking. To that end, we randomly sample $n = 10$ values and evaluate all variants of (6.6). For each result, we explicitly evaluate obtained absolute errors as given in (6.7). We repeat the experiment $10,000$ times to estimate the probability density function of the achieved errors. As inputs, we decided to use independent and identically distributed samples following a normal Gaussian distribution. Figure 6.1 shows the input and output distributions. In our experiments, we used $Q_I = Q_O = Q_A$ the same quantization steps based on the types $T_{3,5}$ and $T_{3,4}$ where the latter is one bit less accurate in the mantissa bit-width. Figure 6.2 shows the probability density function of the achieved errors for both cases. As expected,

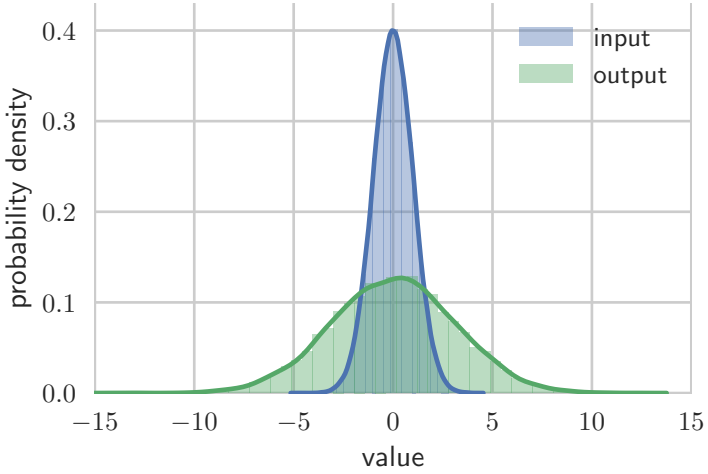


Figure 6.1: Input and output distribution of values. The input is randomly generated by drawing 10'000 samples i.i.d. following a normal distribution.

the error behavior is degraded as more double-rounding occurs when comparing effects from output quantization, extrinsic, and intrinsic approaches. Comparing the top and bottom of Figure 6.2, we observe that adding one additional bit inside the representations improves error patterns by a factor of two. As claimed in (6.8) the upper bound of the error for the intrinsic approach with zero guard bits is equivalent to the extrinsic approach with a Δ_I twice as large as it's original value, in other words, using a data type with one bit less in its mantissa representation. Even though the relations in (6.8) affect the upper bounds, the empirical obtained results depicted in Figure 6.3 demonstrate that the error patterns coincide such that the relation is well maintained including all double-rounding effects measured on data without relying on potentially too over pessimistic upper bounds. This inside suggests using extrinsic evaluations to understand the numerical behavior of deep learning models.

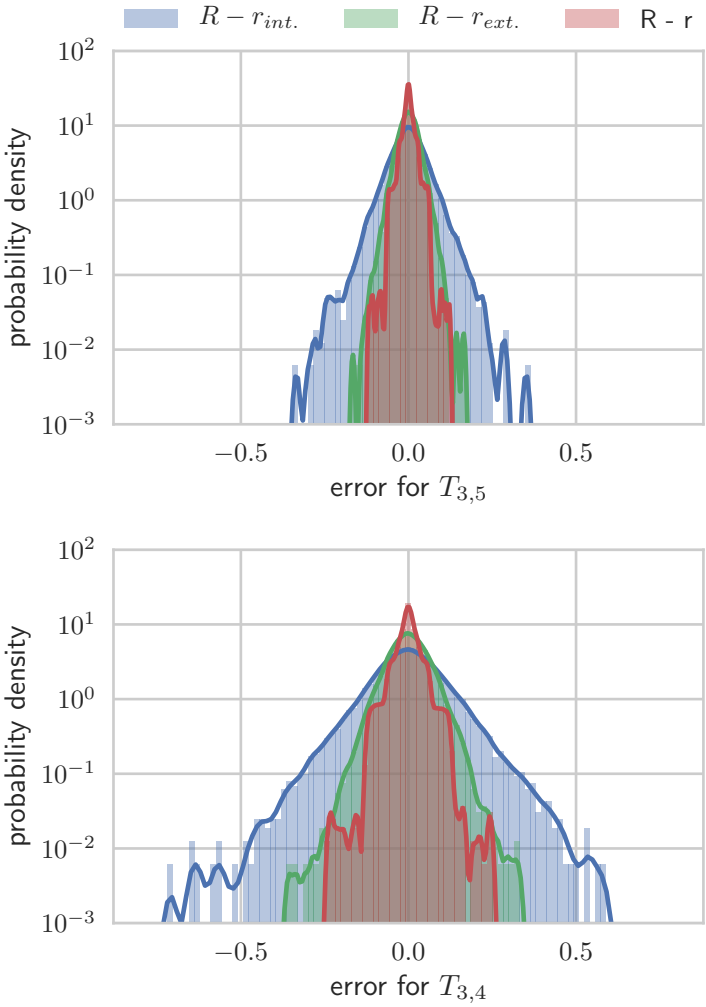


Figure 6.2: Distribution of absolute errors of the intrinsic, extrinsic, and output quantization approach using the same quantization levels $Q_I = Q_O = Q_A$, left for the data type $T_{3,5}$ and right for the data type $T_{3,4}$ with one bit less precision in the mantissa field width.

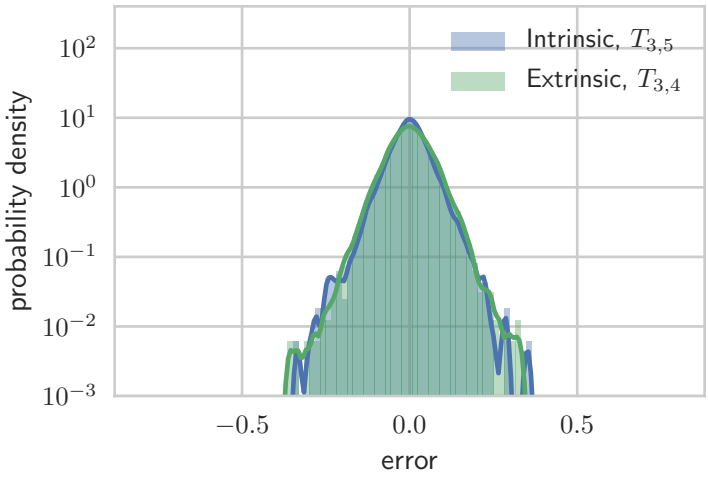


Figure 6.3: The error distribution of the intrinsic approach with input quantization and accumulation quantization of data type $T_{3,5}$ coincides with the extrinsic quantization approach of a one bit less precise evaluation of type $T_{3,4}$.

6.2 Numerical analysis of deep learning models

This section presents the strengths of transprecision computing by demonstrating that a) the concepts are general, b) can be easily integrated into new models, and c) the concepts are scalable. Even though we have demonstrated in Section 5.2.2 that reduced precision representations achieve good results, initial results were obtained by manually introducing the concepts into a reference C implementation of a deep learning model. Even though that intrinsic consideration allowed an in-depth study with full control of low-level details, it still requires manual working steps. In this section, we aim to demonstrate the generality of transprecision computing by applying it to many well-established reference models. We directly operate on the

Table 6.3: Established reference network architectures.

Family	Variant	max batch size	Instances
ResNet	18, 34, 50, 101, 152	1024-128	5
PreActResNet	18, 34, 50, 101, 152	1024-128	5
ResNeXt29	2x64d, 4x64d, 32x4d	256-64	3
DenseNet	121, 161, 169, 201	256-128	4
LeNet	-	1024	1
GoogLeNet	-	256	1
MobileNet	-	1024	1
MobileNetv2	-	512	1
PNASNet	Type A, Type B	1024, 512	2
DPN	26, 92	512, 128	2
SENet18	-	1024	1
VGG	11, 13, 16, 19	1024	4
Total			30

computational graph to avoid for new models to manually write or extend crucial sections of the source code. The pragmatic decision suggests using the extrinsic evaluation policy to avoid writing low-level intrinsic code. To that end, we wrote a utility source code that traverses any computational graph and introduces a parameterized input quantization step for each input of all kernels of the model. We decide to simultaneously study the weight and activation compression of 30 well-established reference models.

6.2.1 Reference models

We decided to use a predefined list of 30 well-established network architectures and report achieved accuracies on CIFAR10. Table 6.3 summarizes the network models we consider as established reference models. Most of them are provided with family-specific and topology related hyper-parameters, such as, for example, visual geometry group (VGG) or ResNets where the parameter refers to the total amount of layers and controls the overall complexity of the model. Figure 6.4 shows averaged normalized execution times for one batch of size 128

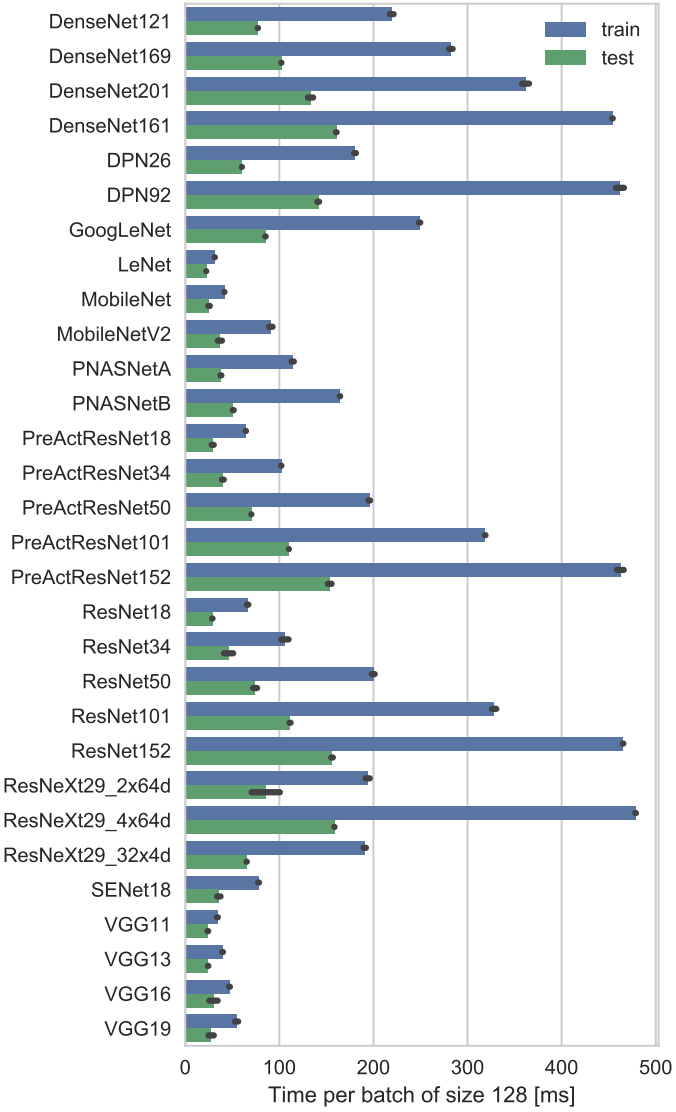


Figure 6.4: Average timings per batch of size 128 for training and testing of reference models. Timings span from 30 ms up to 480 ms per training batch.

images for training and testing. Timings are obtained with PyTorch version 0.4.1 and by running on an IBM Power8 equipped with a P100 GPU. Timings are measured in a realistic setting, e.g., including occurring overheads of loading and transferring data between CPU and GPU and kernel lunch overheads. Training times of one batch include operations caused by backpropagation and the weight update, while testing times refer to the elapsed time the model required to perform a batched forward inference. The fastest model considered is LeNet that trains within 30ms per batch which is 16 times faster than ResNeXt29-4x64d that takes about 480ms per training batch.

6.2.2 Numerical analysis

In this study, we explore the global effect of the number format where one single data type is applied to the entire model. We evaluated a full grid search over 184 floatx configurations for all model topologies. Each configuration corresponds to the reduced precision data type of format $T_{w,t}$ consisting of $w \in [1, 8]$ exponent bits and $t \in [1, 23]$ mantissa bits. Globally applying the same precision configuration enables us to explore the full solution space with a brute-force approach. The knowledge of the full behavior allows to directly answer optimization problems optimally. Details on how to extend and to accelerate the configuration search is further detailed in Chapter 7.

We evaluated all transprecision configurations, on all models, on all 10,000 validation samples of the CIFAR10 image classification dataset. We addressed three questions: Firstly, given a relative quality constraint, what is the best quantization configuration that reduces the bit-width subject to satisfying quality? Secondly, what is the obtained accuracy for a set of configurations of interest? Thirdly, what is the optimal choice of splitting exponent and mantissa bit-widths for a fixed word size of 16 and 8-bits?

Figure 6.5 shows the trade-off between the number format bit-width and the obtained accuracies. Optimal configurations are selected from the full grid search based on satisfying the quality constraint $q(T_{w,t}) \geq Q$ where the requested quality Q is defined relative to the obtained accuracy of the full precision model operating with IEEE 754 32-bit formats as $Q := Q_{float} - \Delta Q$. In this section, quality is measured as Top1 accuracy which corresponds to the total amount

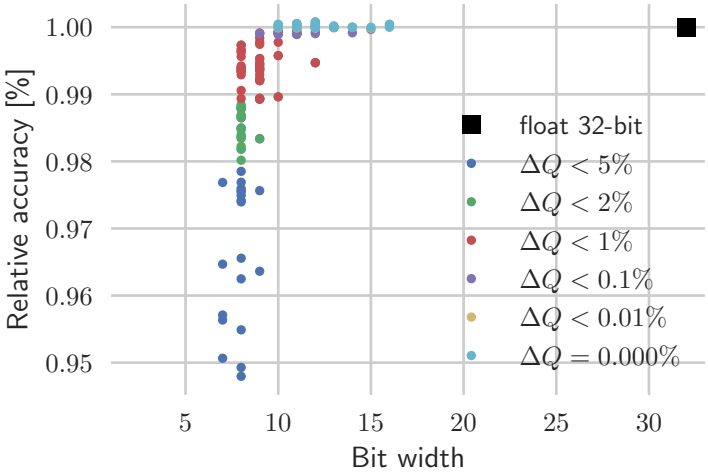


Figure 6.5: Quality versus number format cost in terms of bit-width of the considered reference models. We state the relative accuracy comparing the quantized models against the floating-point 32-bit reference model. Most of the obtained results fall between 8 and 16-bit. On one side, without degenerating accuracy, widths can be compressed with a factor of 2. On the other side, using number formats below 8-bit causes models to fail.

of correctly classified images out the 10,000 validation samples. The results demonstrate that findings generalize well among the variation of network topologies. The strictest constraint of zero-quality-loss—that requires to classifying all out of the 10,000 samples as in the full precision case—is satisfied by reduced transprecision formats. The weaker the constraints, the more aggressive reductions are achievable. Table 6.4 shows best, average, and the worst widths required for different quality constraints for all reference models. On average 12.4-bits are enough to obtain fully accurate results. The bit-width can be further reduced to 8.8 and 8.1-bit if one, and up to five percentage points of quality reductions are allowed. Figure 6.6 shows the

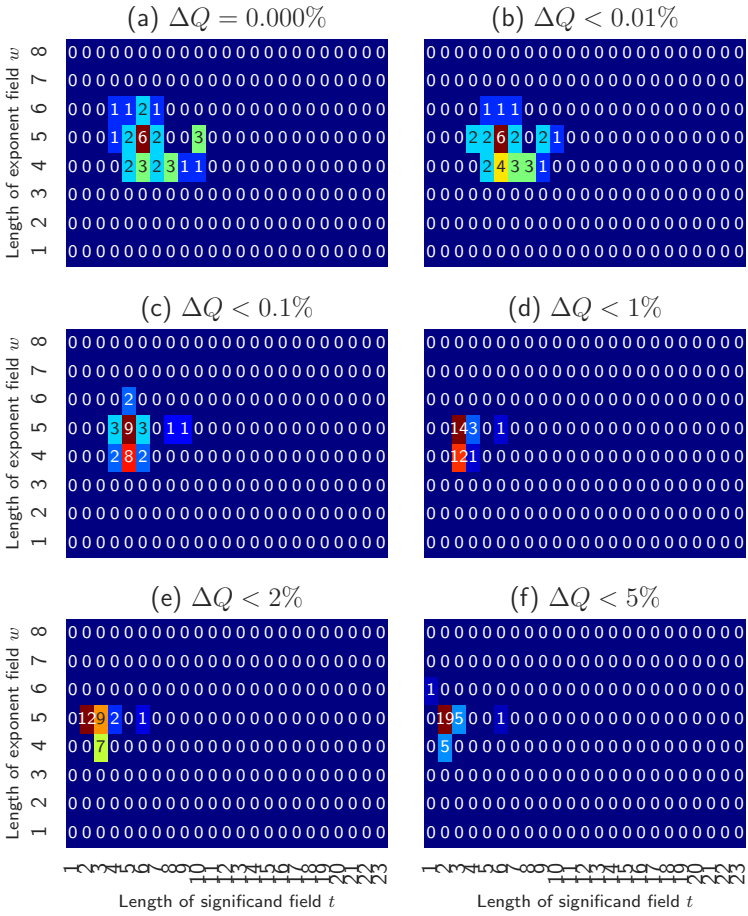


Figure 6.6: Data type configurations for different quality constraints of the considered reference models. Too low and too high exponent field widths are not required. Allowing for larger quality margins against the reference allows to further reduce the mantissa field.

Table 6.4: Statistic of required bit-widths to achieve a given accuracy of the established reference models.

ΔQ	$= 0\%$	$< 0.01\%$	$< 0.1\%$	$< 1\%$	$< 2\%$	$< 5\%$
best	10	10	9	8	8	7
average	12.375	12.125	10.844	8.781	8.531	8.094
worst	16	16	15	12	12	12

configurations of the optimal number formats for the different quality constraints. All results are achieved with a medium-sized exponent field in the range of 4 to 6-bit. The mantissa width ranges between 5 and 10-bit to achieve equivalent results as in the reference. That range is reduced down to 1 to 6-bits if quality degenerations up to 5% points are allowed.

To complete our analysis, we evaluate accuracies for fixed formats that are of particular interest. We optimize the trade-off between exponent and mantissa bits for a fixed word width. We consider the IEEE 754 half format since it is supported on several computing systems, including recent GPUs. Additionally, we consider an alternative encoding of a 16-bit wide format that uses 8 exponent bits to encode the same dynamic range as the 32-bit standard float type and a specific instance of a mini-float 8-bit wide format. The interest of those configurations is motivated by the fact that 8-bit and 16-bit allow directly for four- and two-way vectorization. Data fits into 32-bit words and the parallel ultra-low-power (PULP) platform supports those formats with hardware units and specialized instruction set extensions.

Figure 6.7 shows the accuracy loss against the floating-point 32-bit reference for the three selected formats of all networks. On average, the IEEE 754 half format leads to an accuracy degeneration of 0.006 percentage points and the 16alt lowers the accuracy by 0.028 percentage points. The half format works better in most of the cases. The reason behind that observation is that in almost all cases 4 or 5 bit are enough to cover the required dynamic range of the weights and internal activations. The additional extended dynamic range of the 16alt format does not help to improve results. However, in

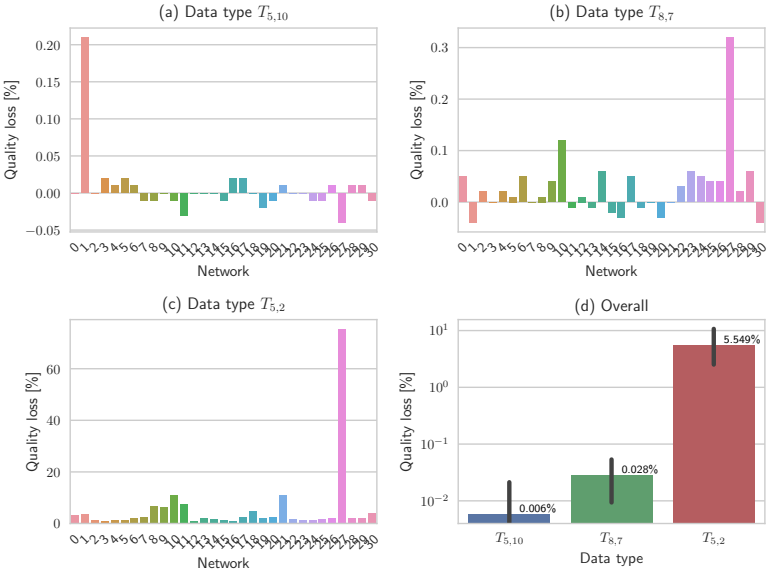


Figure 6.7: Performance of selected formats, IEEE 754 half, 16alt, a the mini 8-bit format $T_{5,2}$ measured against the accuracy of the float 32-bit reference. Negative numbers indicated that (due noise) a better performance of the quantized models. The first three figures depict results per network architecture, the last figure summarizes the results.

contrast, the additional three mantissa bits of half improve results. To study trade-offs between exponent and mantissa, we evaluated all fixed bit-width combinations up to 8 exponent bits. Figure 6.8 shows results measured as average accuracy loss compared against the full precision reference. Reduced dynamic range causes the model to fail with highly degenerated results. In contrast, a too-large exponent extends the dynamic range beyond the space that is required during computation causing a lower quality due to a reduced mantissa field. That study reveals, that the IEEE 754 half standard width 5 exponent and 10 mantissa bits is close to optimal and achieves a regret of

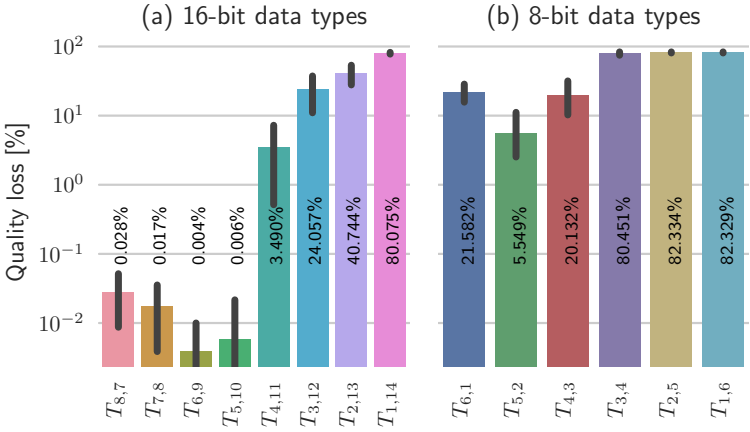


Figure 6.8: Performance of (a) 16-bit and (b) 8-bit formats measured against the accuracy of the float 32-bit reference. Computations with formats of four or less exponent bits cause high degenerations. Large exponent widths correspond to smaller mantissa widths which slightly degenerate quality once the format spans the dynamic range.

0.006%. That result is outperformed by $T_{6,9}$ which yields a regret of 0.004%. Considering 8-bit formats, the optimal choice is $T_{5,2}$ that yields a regret of 5.5%, however, that value is mostly determined by one single outlier, the median is 2.3% point. In this respect, the data type $T_{5,2}$ supports the argument that mini float8 formats should have 5 exponent bits. This agrees with the design choice for mini-float8 implementations supported on the PULP system. The layout of $T_{5,2}$ is motivated by the fact the number of exponent bits cover the same dynamic range as in the IEEE 754 half type.

6.3 Summary and conclusion

The most important findings of this chapter are the following:

- Deep learning is a fast-evolving field causing various interesting research and engineering subfields. We decided to focus on pixel-data for the problem of image and video classification. Working with pixel data provides a challenging use case with high dimensional input spaces and large volumes of data. Over the past decade, GPU peak performance increased from 1TFLOPs to well above 10 TFLOPs while the accuracy on CIFAR10 moved from 80% to above 98%. Within the last three years, the average cited accuracy on CIFAR10 was improved by 1.3% point per year.

- In Section 6 we integrate floatx into PyTorch to power transprecision computing analysis and evaluation on model level. We discuss and relate intrinsic and extrinsic emulation approaches and conclude that the intrinsic approach is much fast to compute and emulates well-enough the numerical behavior with simpler source code. We evaluated a list of over 30 well-established reference models and support the finding that deep learning models are error-resilient and well-suited candidates for transprecision computing. We evaluated all configurations of floatx number formats and conclude that number representations can be compressed by 4× within a less than 5% point quality degeneration. Even in the strictest case of achieving a zero-loss solution, the number representations can still be compressed by 2.6× on average.

Within this work we focused on pixel-data, however, we state that the developed concepts and results are general and we do not see conceptual limiting factors that would hinder future work to apply the develop principles to a broader context of different deep learning use-cases including different data, such as audio, text, or use case-specific data, or different tasks such as object recognition, regression, or forecasting. We hope that the error resilience property is further exploited and integrated into more domains.

Chapter 7

Optimization for transprecision configurations

In the previous chapters, we discussed the fundamental concepts of transprecision computing which allow trade-offs between quality and performance of applications. Several aspects, such as benchmarking, using floatx to emulate numerical effects, and integrating floatx into PyTorch [62] was investigated. Moreover, we demonstrated the scaling of the concepts on various deep learning models. In presented cases, the cardinality of the configuration space Θ was limited such that it was affordable to evaluate all operational points of interest. We used case-specific assumptions to simplify optimization problems. We arrived with small enough, finite-lists of alternatives that make the optimization problem (4.8) trivial. However, large configuration spaces are essential for the success of transprecision computing. Transprecision computing must demonstrate the usefulness at a broad spectrum including scalability, affordability, and successful; inspections of non-trivial configuration spaces. Two main aspects are essential to achieve success: (1) transprecision computing must be *easy-to-apply* and (2) it must be *fast-to-deploy*.

Transprecision related and first-order system parameters have a joint influence on quality and performance characteristics. Simultaneously understanding both allows judging the value of opportunity costs by comparing transprecision variants with different system configurations. In that context, the key-aspects *easy-to-consume* and *fast-to-deploy* need to demonstrate success in the broad field of alternative solutions. We discuss how to limit the combinatorial grow of the search space cardinality with effective strategies in Section 7.1. Section 7.2 proposes heuristic search techniques to quickly solve the configuration problem. We present in-depth results on a reference problem in Section 7.3 and we extend the search to over 30 established models in Section 7.4.

7.1 Searching transprecision configurations

We consider the case where a traditional computer program that is composed of n variables $x_1, x_2, x_3, ..., x_n$ which are represented by data types $T^1, T^2, T^3, ..., T^n$. In traditional computing, where no mixed data types are used, all types are equivalent. For example $T^1, T^2, T^3, ..., T^n = T_{5,23}$ are all of type IEEE 754 32-bit floating-point. However, when using the concept of transprecision computing, various data types might be used instead. The choice of which datatype should be used per variable, naturally leads to a configuration search problem. Considering targeting current hardware, that only implements a small set of data types, simplifies the search problem. Assuming the availability of a transprecision system that supports various data types of fine granularity, the number of choices grows and the optimization becomes a challenging and time-consuming problem. The number of valid configurations of an optimization problem grows exponentially in the number of considered variables, leading to a substantial search space even when considering a reasonable number of variables.

In this section we consider the transprecision search problem for floatx as introduced in Section 5. We study reduced precision types $T_{w,t}$ for $w \in [1, 8]$ and $t \in [1, 23]$ that consider the search problem up to the standard float 32-bit representation. The bit-level granularity leads to a total of 184 $(8 \cdot 23)$ options as a choice of type for each of

the n variables. Finally, we formulate the search problem as finding the best transprecision configuration $\theta^* \in \Theta$ meeting performance or quality constraints as in (4.8) and (4.9). Since there only exist a finite set of precision configurations, a brute force approach solves the optimization problem. However, the amount of configurations evaluates to 184^n and renders such an approach intractable, even for small numbers of n. For example, for $n = 10$ variables, the search space cardinality exceeds $4.5 \cdot 10^{22}$. In this context, we study systematic methodologies to reduce the search effort by defining more traceable problem instances that still provide application insights.

Global search

The global search imposes the assumption that all data types used in a program are equivalent to a single global type: $T_{global} = T^1 = T^2 = T^3 = ... = T^n$. This approach reduces the space to the case of $n = 1$ and renders a full grid search manageable. Even though the global datatype assumption sounds very restrictive, there are several arguments that motivate such an approach. First, the global search provides insights into the error resilience of applications. For example, global search results can act as lower bound to subsequent searches performed at finer granularity. Second, there is no need of performing casts between different data types. Casting between different formats might require a change to the underlying memory footprint or to spend computing time to perform the required operations. In that case, potential gains due to reduced precision must be large enough to amortize required cast overheads. Third, in FPGA or ASIC designs one global format ensures full design freedom for resource sharing.

Sensitivity analysis

The sensitivity analysis formulates a search problem by elaborating the effect of reduced precision of each variable in the system independently of the others. To that end, the type T^i of the i-th variable is swept through the full grid while all other variables are kept at full precision type, e.g., $T^j = T_{5,23}$ for $j \neq i$. The sensitivity analysis costs one full gird evaluation (e.g., evaluating 184 configurations) for one single variable and is repeated n times for each variable, leading

to a total amount of $184 \cdot n$ evaluations. The linear dependence on the number of variables makes this approach scalable. The sensitivity analysis provides an easily interpretable result that can be used to identify precision bottlenecks.

Grouping

Alternatively, grouping variables together into sets reduces the number of configurations in the solution space. One data type is applied to all variables in the group. Generally, we define g groups $G_1, G_2, G_3, ..., G_g$ that contain the variables belonging to the defined group $G_i := i_1, i_2, i_3, ..., i_k$. By definition, groups are disjoint $G_i \cap G_j = \emptyset$ for $i \neq j$ and the union covers all variables $G_1 \cup G_2 \cup ... \cup G_g = 1, 2, 3, ..., n$. The group configuration itself is a required input and the choice might be influenced by reasonable side constraints. For example, for a program of medium size, it makes sense to group together variables that belong to the same code fragment. For each group, only one dedicated data type is assumed $T^{i_1} = T^{i_2} ... = T^{i_k}$ that is shared among all variables of this group. Such a consideration leads do reductions whenever $g < n$ and reduces the search space cardinality from 184^n to 184^g.

7.2 Search heuristics

In this section we study the effects of using heuristics that allow the further shortening of the evaluation time. A full grid search over one variable is of cost O(number of TP configurations) $\cdot$ O(evaluation effort per configuration). We define heuristic searches for finding good—but not guaranteed to be optimal—transprecision configurations with much less effort. As introduced in Section 4.2.1 we denote the final application quality as Q. We formulate the assumption that an increase in Q stems from a wider number representation, either a larger exponent field or a more precise represented mantissa:

$$Q(T_{w_1,t_1}) \leq Q(T_{w_2,t_2}) \quad \text{if} \quad w_2 \geq w_1 \quad \text{and} \quad t_2 \geq t_1. \tag{7.1}$$

If we know a priori that (7.1) holds, it directly enables efficient implementations. For example, exponent and mantissa field-widths

can be searched with binary search. However, we know from prior micro-benchmarks that inequalities do not strictly hold in all cases. Depending on the application, different quantization-based noise patterns occur and lead to local anomalies. We propose several search heuristics and evaluate their performances in a controlled setting.

Smart region search pattern with early exit

Instead of computing the full grid followed by selecting the best candidate, it is more efficient to evaluate configurations of small data types first. This enables an early exit, since when the quality constraint $Q(T_{w,t}) > Q_{target}$ is met, we can assure that the data type $T_{w,t}$ is optimal. Since by construction, all smaller data types have already been searched for and do not meet the quality constraint. The worst-case execution time is the same as performing the regular full grid search. However, the early success of finding candidates enables a reduction in the average search time. The search sequence is simple: according to the minimization target $P(T_{w,t}) = 1 + w + t$, short bit-width formats are preferred. Among configurations with the same bit-width, we prefer the transprecision configuration with the larger exponent. This search construction leads to a diagonal based pattern in the configuration matrix. The resulting search sequence is $T_{1,1}$, $T_{2,1}$, $T_{1,2}$, $T_{3,1}$, $T_{2,2}$, $T_{1,3}$, ..., $T_{8,23}$. Additionally, to the diagonal search pattern, the same idea can be applied to search any regions of interest efficiently. Let us assume that an arbitrary region of interest is given and defined as a Boolean mask over the configuration matrix. In that case, the same diagonal search pattern can be applied with the minor modification that for each configuration a check is performed if it belongs to the region of interest or not. Only valid marked configurations are required to be evaluated. This variant of the smart region search pattern with an early exit is relevant as sub-step in the coarse-to-fine heuristic described in Section 7.2

Parallel exponent/mantissa binary search

Modifying the bit-widths of exponent and mantissa results in several effects. In both cases, we expect that an increase in bit-width leads to a monotonic increase in quality. Changing the mantissa is equivalent

to modifying the precision of local computations. Its impact is well understood for relevant problem classes and is for example covered with perturbation theory for linear operations. Changing the exponent limits the range of representable values, the effect is highly non-linear and typically leads to a breakdown of the computation if internal under- or overflows occur. However, predicting the full application behaviour is challenging since effects depend on input data and the application-specific chains along which errors propagate. Changing exponent and mantissa configurations simultaneously depends on the above as well as on cross effects of the joint interactions. To that end, we can split the quality assumptions into exponent and mantissa related formulations:

$$Q(T_{w_1,t}) \leq Q(T_{w_2,t}) \quad \text{if} \quad w_2 \geq w_1 \text{ and } t \text{ constant} \tag{7.2}$$

$$Q(T_{w,t_1}) \leq Q(T_{w,t_2}) \quad \text{if} \quad w \text{ fix and } t_2 \geq t_1 \tag{7.3}$$

The separated assumptions are weaker than the joint assumption since they are not required to make any assumptions about cross-effects of simultaneous changes on exponent and mantissa widths. Both formulations lead directly to efficient implementations that we name parallel exponent binary search and parallel mantissa binary search. In the first case, we perform for each $t \in [1, 23]$ an independent binary search that operates over the exponent range. Similarly, in the second we operate independent binary searches over the mantissa for all fixed values $w \in [1, 8]$ of the exponent. In terms of performance, the parallel exponent binary search is of order $O(log_2(8) * 23)$ and the parallel mantissa binary search is of order $O(8 * log_2(23))$. This way, the later provides a better computing performance and in the case of a monotonic behaviour of the quality based on the mantissa should also algorithmically work better than a search based on the exponent.

Two-stage search

To further reduce the number of evaluations, we propose algorithms that reduce the evaluation effort simultaneously for exponent and mantissa. To that end, we suggest a two-stage approach where we apply two binary searches in sequence, where the latter is started

over a column or row found by the first search. The two-stage search operates as follows:

- Input: pivot for the first search,
- Step 1: run a binary search at the pivot index,
- Step 2: run the second binary search, starting with results obtained from step 1.

Two-stage searches are more aggressive in reducing the number of evaluations and are henceforth expected to return slightly suboptimal results. However, for rapid evaluation, such an approach renders useful. For example, many applications obey a typical error pattern with distinct behaviors for mantissa and exponent where sharp transitions between high and low quality are especially present based on the exponent length. Within these two regions, smoother transitions or noise based on the mantissa changes might be observed. BLSTM is one benchmark we presented in previous deliverables that follows such an error pattern. In such cases, typical pass-fail regions build rectangular shaped regions. Therefore, it is enough to first search for the transition on the boarder of the search space and then search towards the inner region of the search space. We formulated the search algorithm in a way that a pivot row or column is defined where the first search is performed and the second search follows based on the results of the first search. We consider the pivot row to be either at the boarder (low or high) or in the middle. Even though we are fully aware that different input configurations lead to different results, we evaluate all six configurations and we discuss in which direction results tend to be biased in the result section.

Coarse versus fine search with adaptive evaluation effort

In Section 7.2 we studied the effect of heuristics that reduced the number of configuration evaluations. Those considerations assume a constant time effort for evaluating one configuration. In some applications, however, an approximation of the quality can be computed faster. For example, assessing the quality of machine learning algorithms requires to compute the accuracy by feeding test data through

a model. In that case, using a subset of samples reduces the evaluation effort for one transprecision configuration. Evaluating models in a controlled environment includes computing quality numbers with the full amount of provided samples. To that end, subsampling the test set is not adequate. However, in the configuration search case, we know that we potentially asses hundreds of configurations whereas only the last configuration is required to be evaluated with the full effort. Prior decisions can be based on approximated quality estimates. Following this method, we formulate the fast version of the evaluation routine and build the search configuration heuristic around it. For evaluating subsets of data define the following:

- Number of samples
- Sample selection procedure
- Quality metric computation based on splits

The evaluation effort depends linearly on the number of samples. However, there are two limitations identified. First, sample sizes smaller than the typical batch-size do not scale any more. Second, the number of samples in the subset should at least cover all existing classes.

Next, we assume general selection where samples are random uniformly selected among all samples in the test set. Since the approximated quality metric contains distortions caused by the random selection, we estimate confidence intervals. To that end, we split the subset into k equally sized splits to compute mean and variance over the computed coarse-grained metrics. More formally, we compute the pair (μ, σ) based on the evaluation configuration (s, k) where s denotes the total number of samples in the subset and k denotes the number of splits.

Coarse-to-fine search:

- Step 1: perform a coarse full grid search with effort (s_{coarse}, k_{course}).
- Step 2: filter configurations into {good, undecided, bad}.
- Step 3: perform a 1st fine search in the {good or undecided} region.

- Step 4: if more effort is desired, perform a 2nd fine search the undecided or good region.

The reason for using heuristics is to rule out with little effort unusable configurations while keeping the most promising configurations. In the first step, the full grid is scanned with low effort. In the second step, the algorithm decides what happens with a configuration that produced a result (μ, σ) according to the following rule:

$$f(\mu,\sigma) = \begin{cases} good & \text{if} \quad \mu - \max\left(\alpha_{good}\sigma, \epsilon_{good}\right) \geq \tau \\ undecided & \text{else} \\ bad & \text{if} \quad \mu + \max\left(\alpha_{bad}\sigma, \epsilon_{bad}\right) \leq \tau \end{cases} \tag{7.4}$$

The filtering detects good and bad configurations easily and enables to focus on the undecided candidates. Four parameters characterize the filtering step. The factors α_{good} and α_{bad} define noise levels relative to the standard deviation. If the mean of the approximated quality level sufficiently exceeds or undershoots the limit, the configuration is either considered good or bad. The parameters ϵ_{good} and ϵ_{bad} specify minimal noise levels around the mean that have to be respected. In steps 3 and 4 the classified configurations are further inspected by using the smart region search pattern with early exit as described in Section 7.2. Two parameters define the exact behaviors, one defines which of the good or undecided classified configurations should be considered in the first fine search and the second flag defines if the remaining class should be search for as well or if it should be skipped. Filtering parameters affect the classification, the triggered search effort, and the result quality. We observed that two configurations are appropriate. First, to optimize search performance, we directly search the good candidates and skip the undecided candidates. Since good candidates passed a coarse quality check, the chances are high that the first considered best configuration will satisfy the constraint in the fine-grained evaluation. That way, the algorithm quickly terminates after a few fine-grained evaluations. Second, for better-tuned results, undecided configurations are explored first. The heuristic terminates with success, if a candidate that respects the quality constraint is found. However, since it is not known how likely

it is to actually find a valid configuration, the search might be forced to evaluate all undecided configurations without success. This effect increases the average runtime of this variant of the heuristic search. To overcome a fail, it pays-off triggering the 2nd fine search over the remaining good configurations to quickly find a good alternative candidate.

7.3 Results on reference problem instance

We demonstrate the proposed heuristic algorithms on image classification with ResNet18 [24] on CIFAR10 [22]. We solve the global search problem, where quantization is applied to all model parameters and to all intermediate activation functions passed between operations. We quantize models a posteriori without retraining.

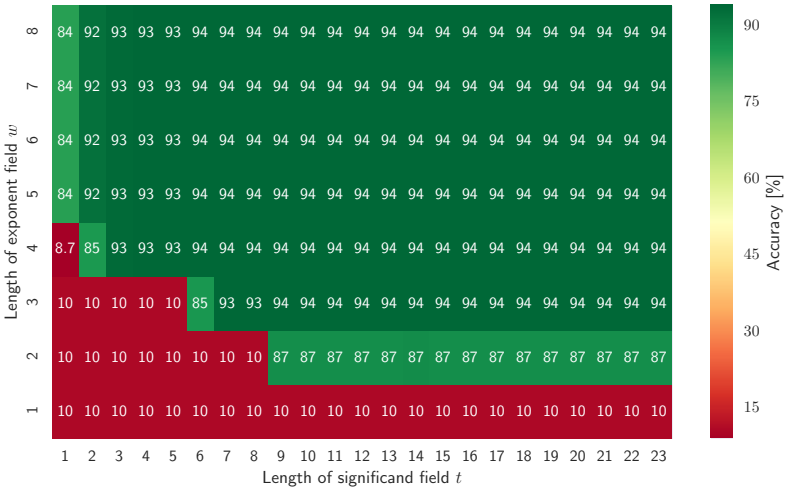


Figure 7.1: Full grid search results where all valid floatx configurations are exhaustively evaluated on the reference problem instance (floatx quantized ResNet18 on CIFAR10). Fail/pass regions are sharply separated and increasing exponent or mantissa field width improves the accuracy.

Table 7.1: Selected operation points for different quality levels on the reference problem instance.

ΔQ	Q^1_{target}	Optimal format	Bit-width	Accuracy
0.000%	93.58%	$(4, 10)$	15	93.58%
0.01%	93.57%	$(4, 8)$	13	93.57%
0.1%	93.48%	$(4, 6)$	11	93.52%
1%	92.58%	$(4, 3)$	8	93.04%
2%	91.58%	$(4, 3)$	8	93.04%
5%	88.58%	$(4, 3)$	8	93.04%
10%	83.58%	$(4, 2)$	7	84.87%

[1] Q_{target} is computed as $Q_{float} - \Delta Q$.

Figure 7.1 shows the obtained quality for an exhaustive grid search over the 184 configurations. Similarly, as in previously reported results (for example BLSTM, see Section 5.2.2), we identify a clear pass/fail region. The floating-point 32-bit accuracy of the model is $Q_{float} = 93.58\%$. If we request a quality of $Q = 93.00\%$ the optimal number format is $T_{4,3}$ with an accuracy of 93.04%. Table 7.1 shows different operation points. We illustrate the proposed variants of heuristics by solving the reference problem instance with a requested quality of $Q = 93.0\%$ in the next sections.

Smart region search pattern with early exit

The smart region search algorithm guarantees to yield the optimal result but is required to search the full grid in the worst-case. The algorithm follows a diagonal search pattern that prefers small formats over larger formats. It terminates directly whenever a configuration is found that meets the quality constraint $Q(T_{w,t}) \geq Q_{target}$. Figure 7.2 illustrates the situation for the reference problem instance. In this case, 18 evaluations were enough to find the optimal configuration $T_{4,3}$.

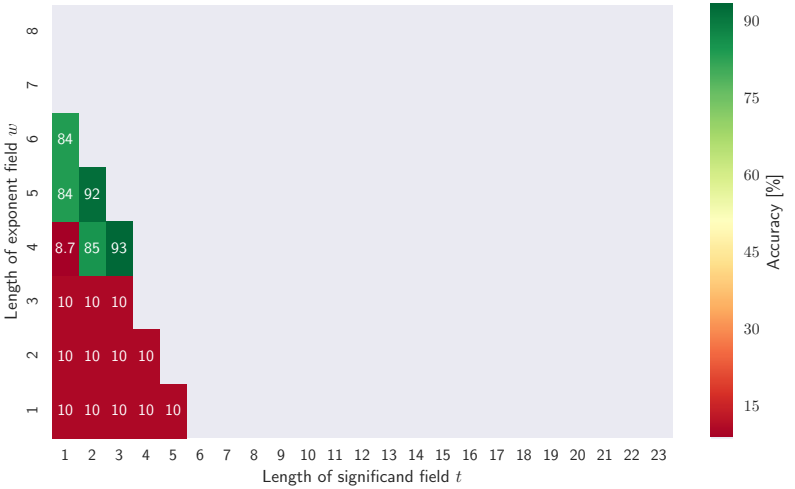
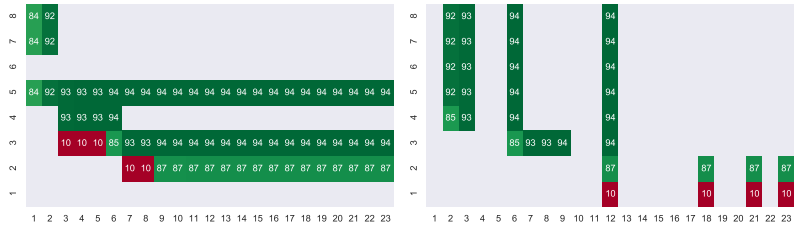


Figure 7.2: Illustrative run of the smart search on the reference problem. The region is searched following a diagonal pattern from the left-bottom corner and quickly terminates at the optimal configuration.

Parallel exponent/mantissa binary search

Figure 7.3 shows the search patterns for the parallel binary search on the reference problem. The on the left Subfigure the binary search is performed on the exponent, on the right Subfigure on the Mantissa. In both cases, the heuristic approach is able to find the global optimal solution. Working with the binary search in exponent direction leads to a total of 69 evaluation calls. However, applying binary search in mantissa direction enables higher performance gains since the logarithmic scaling is applied to the larger factor. In total 33 evaluations are necessary to converge to the same result in this case. Parallel exponent and mantissa binary searches reduce the search effort by 2.6×, and 5.5×, respectively.



(a) parallel exponent binary search (b) parallel mantissa binary search

Figure 7.3: Illustrative run of the parallel exponent/mantissa binary search. Both searches found the optimal solution.

Two-stage search

Figure 7.4 shows the search patterns for the two-stage binary search on the reference problem. In this specific example, choosing the pivot element too small is problematic since in that case the first search passes over large amounts of the failed region of the configurations and does not provide useful hints of where good configurations can be found. The second search is therefore triggered at the worst-case assumption of the first search (i.e., largest exponent or mantissa). In both cases, configurations that strictly satisfy the quality constraint are still found, however with a suboptimal transprecision configuration that consists of larger than optimal bit-width. Using a medium or large pivot as initialization performs equally well and converges in this problem instance to the same solution. In the case of first searching the exponent, too optimistic values of for the exponent fields ($e = 3$) are founded first and henceforth the second search leads to a suboptimal solution. In the case of first searching the mantissa followed by searching the exponent afterward leads to the global optimal solution in this problem instance. In this case, 7 configurations are triggered for evaluation leading to an over 26× speedup in the search.

Coarse versus fine search with adaptive evaluation

We characterize the effect of reduced effort evaluations by repeating the full grid search on subsets of the test set. We selected a subset size

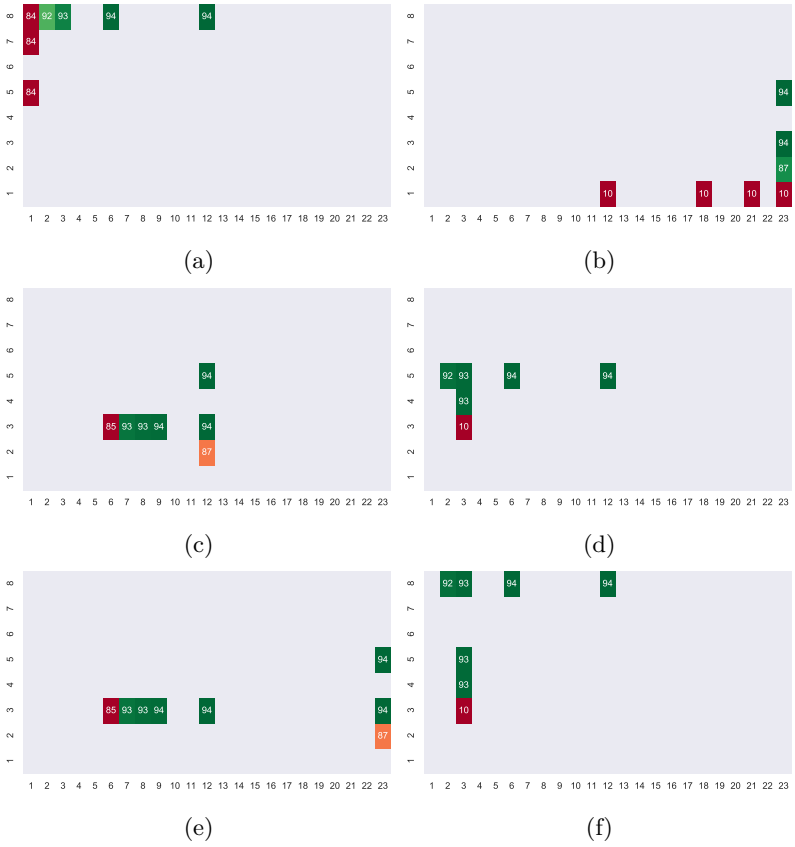


Figure 7.4: Illustrative runs of the two stage searches. The first search is performed in exponent (left) and mantissa (right) directions. The pivot is low, medium, and high for the tree rows of figures.

of 1%, 10%, and 100% of the total number of samples. Each subset if further split into $k = 5$ splits to evaluate k independent accuracy results. Those results are merged together by computing the mean and standard deviation. Figure 7.5 shows the achieved results. Even using a very small subset of 1% size, the overall character of fail/pass regions are preserved. However, the accuracy of the subset might over or underestimate the real accuracy. The standard deviations indicate how the uncertainness is reduced when enlarging the sample size, the standard deviation achieves values of about 2%, 1.3%, and converges at around 0.5% points.

Based on that observation, we decide to parametrize the coarse to fine algorithm with two configurations:

- Default: $(s = 100, k = 5)$, $\alpha_{good/bad} = 1$, $\epsilon_{good/bad} = 1$
- Conservative: $(s = 100, k = 5)$, $\alpha_{good/bad} = 3$, $\epsilon_{good/bad} = 5$

We decide to use $s = 100$ samples to perform the coarse search, a decision that allows to evaluating CIFAR10's validation set $100\times$ faster than the regular evaluation mode where all 10,000 samples are considered. As a default setting, we suggest to filter based on one standard deviation and to enforce at least one percentage point of error guard band. In contrast, to be a bit more conservative, we suggest to use a 3σ filter threshold and to enforce at least five percent point of error guard band.

Figure 7.6 shows the filtering masks for default (left) and conservative (right) settings, where the following encoding is used 1: good, 0 undecided, -1 bad. In both cases, the bad performing region can be safely detected. However, since the requested search quality is close to any good performing network, it is more likely that the requested quality level is inside the error bounds of the estimated accuracy. That causes the conservative setting to classify all configurations towards the left top side as undecided. In both cases, the lower two plots show aggregated results, when the first and second search are triggered on the undecided and the good candidates for the default (left)and conservative (right) choice of the search parameters. Only three and six full-effort evaluations are required to converge to the optimal global solution in this case.

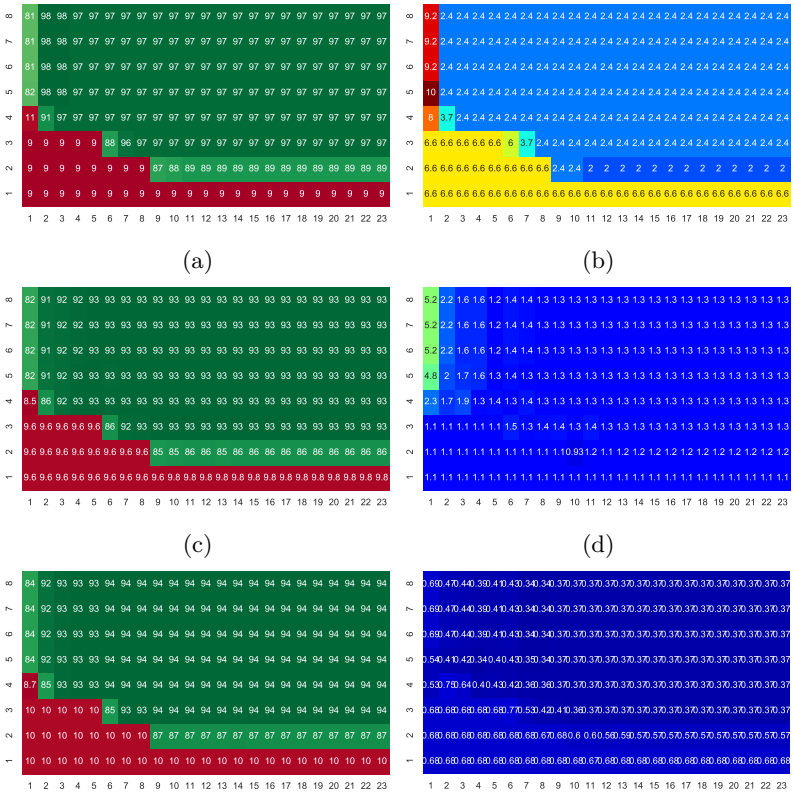


Figure 7.5: Illustrative characterization of reduced effort evaluation. Top-down the figures correspond to an effort of 1%, 10%, and 100%, where left figures show the mean and right figures show the standard deviation over accuracy estimates obtained over five splits.

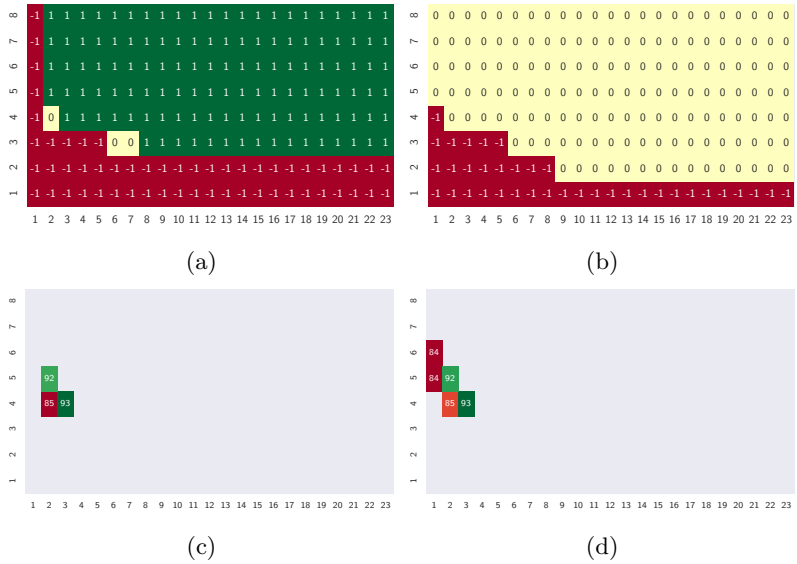


Figure 7.6: Illustrative run of coarse to fine heuristic. Top: filter mask for default (left) and conservative (right) settings. Bottom: triggered fine search evaluations based on the two filtering masks.

7.4 Heuristic search performance

Since the search performance depends directly on the data, we evaluate it over a representative set of problems. To that end, we used the thirty well established reference models presented in Section 6.2.1 for which we have performed a global exhaustive search, such that all optimal results are known. Since each model has a specific best performance, we define the requested search quality relative to it. We measure performance and quality by averaging results over the set of reference models.

Smart region search pattern with early exit

Table 7.2 states achieved performance gains of the heuristic compared against the exhaustive search. The smart search guarantees to find the

Table 7.2: Performance of the smart region search pattern with early exit. The heuristic delivers optimal configurations up to 10× faster.

ΔQ	Q_{target} ($Q_{float} - \Delta Q$)	Avg. $Q(\theta^*)$	Avg. $P(\theta^*)$	Avg. calls	Search speedup
0%	92.341%	92.355%	12.5	51.8	3.5×
0.01%	92.331%	92.351%	12.2	49.9	3.7×
0.1%	92.241%	92.288%	10.9	39.5	4.7×
1%	91.341%	91.807%	8.8	23.3	7.9×
2%	90.341%	91.351%	8.5	21.2	8.7×
5%	87.341%	89.935%	8.1	18.3	10.1×

optimal configuration as already presented in Table 6.4. Henceforth, the average obtained accuracy is greater or equal to the requested quality. Since the search is performed from small to large formats the average search time is longer for higher requested accuracies. On average, between 20 to 50 calls are required to find a configuration that satisfies the constraint. Henceforth, the smart search finished between 10× and 3.5× faster than the full grid search without loss in quality.

Parallel exponent/mantissa binary search

Table 7.3 states results for the exponent/mantissa binary search heuristic. Sometimes, noise causes wrong decisions of the binary search that lead to suboptimal solutions. We measure the quality of the heuristic by compute the absolute quality difference $\Delta q = Q(\theta^*) - Q(\theta^{ref})$ and absolute cost difference $\Delta p = P(\theta^*) - P(\theta^{ref})$ between the obtained solution θ^* of the heuristic compared against the global optimal solution θ^{ref}. Accuracies are almost the same in all cases. For less strict quality requirements $\geq 1\%$ the found results are equal to the optimal solution in all cases. For strict quality requirements, heuristically found results might differ from the optimal solution. However, still the vast majority (at least 16 out all cases) the exact optimal solution is found. For suboptimal results, the required quality constraint is

Table 7.3: Performance of the parallel binary search heuristic.

Dir.	Req.	Avg.	Avg.	Result classification				Avg.	speedup
	ΔQ	Δq	Δp	$=$	$+1$	≥ 2	≥ 5	calls	
Mantissa	0%	0.000	1.06	16	2	13	0	34.6	5.3×
	0.01%	-0.001	0.77	18	3	10	0	34.3	5.4×
	0.1%	0.000	0.10	29	0	2	0	35.8	5.1×
	1%	0.000	0.00	31	0	0	0	33.7	5.5×
	2%	0.000	0.00	31	0	0	0	34.9	5.3×
	5%	0.000	0.00	31	0	0	0	36.4	5.1×
Exponent	0%	-0.001	0.16	28	1	2	0	69.0	2.7×
	0.01%	-0.001	0.16	28	2	1	0	69.0	2.7×
	0.1%	0.000	0.00	31	0	0	0	69.0	2.7×
	1%	0.000	0.00	31	0	0	0	70.4	2.6×
	2%	0.000	0.00	31	0	0	0	72.1	2.6×
	5%	0.000	0.00	31	0	0	0	73.1	2.5×

strictly met in all cases due to the formulation of the algorithm. However, the average bit-width is increased by up to 1.06-bit. The runtime is constant and does not depend on the requested quality. The search over the wider mantissa outperforms the alternative that searches over the exponent. The average speed-up in the former case amounts to 5.3×.

Two-stage search

Table 7.4 states the results for all six configurations for direction and pivot. The search is defined that it finds in all cases a solution that strictly satisfies the user given quality constraint. Comparing the obtained accuracy with the value of the global optimal solution shows that in all relevant cases the results are very similar and differences are below 0.1%. In one case (Mantissa, low) the difference is more than 1.5% point. However, in that case, a more accurate solution was found at the cost of additional 19-bits on average. Choosing the pivot value as low is problematic. Since in most cases results break through a degenerated exponent or mantissa representation, the first search

Table 7.4: Performance of the two-stage binary search heuristic.

Dir.	Piv.	Req. ΔQ	Avg. Δq	Avg. Δp	Result classification =	+1	≥ 2	≥ 5	Avg. calls	speed-up
Mantissa	low	0%	-0.015	15.2	0	0	0	31	7.0	26.3×
		0.01%	-0.011	15.5	0	0	0	31	7.0	26.3×
		0.1%	0.053	16.7	0	0	0	31	7.0	26.3×
		1%	0.448	18.6	0	0	0	31	7.1	25.9×
		2%	0.617	18.7	0	0	0	31	7.2	25.5×
		5%	1.531	19.0	0	0	0	31	7.3	25.2×
Mantissa	mid	0%	-0.001	2.5	14	1	11	5	7.5	24.6×
		0.01%	-0.002	1.5	16	3	10	2	7.4	25.0×
		0.1%	0.007	0.8	25	1	4	1	7.7	23.9×
		1%	0.000	0.0	31	0	0	0	7.1	25.8×
		2%	0.000	0.0	31	0	0	0	7.5	24.7×
		5%	0.122	0.0	30	1	0	0	7.8	23.6×
Mantissa	high	0%	-0.004	2.1	13	1	13	4	7.5	24.6×
		0.01%	-0.004	1.3	14	3	13	1	7.4	24.8×
		0.1%	0.002	0.4	22	3	6	0	7.7	23.9×
		1%	0.045	0.1	29	2	0	0	7.2	25.6×
		2%	-0.040	0.0	30	0	1	0	7.5	24.6×
		5%	0.000	0.0	31	0	0	0	7.8	23.6×
Exponent	low	0%	-0.003	5.4	0	0	27	4	7.5	24.6×
		0.01%	0.001	4.7	0	0	30	1	7.4	24.8×
		0.1%	0.010	3.6	0	0	31	0	7.7	23.9×
		1%	0.113	3.5	0	0	31	0	7.2	25.6×
		2%	-0.008	3.2	0	0	31	0	7.5	24.6×
		5%	0.279	3.1	1	0	30	0	7.8	23.6×
Exponent	mid	0%	-0.003	2.9	8	0	19	4	7.5	24.7×
		0.01%	-0.004	2.4	8	1	19	3	7.3	25.4×
		0.1%	0.004	1.8	8	1	20	2	7.7	23.9×
		1%	0.111	2.5	9	0	15	7	7.7	23.9×
		2%	0.077	3.0	6	1	12	12	7.8	23.7×
		5%	0.036	3.0	6	2	8	15	8.0	23.0×
Exponent	high	0%	-0.005	3.8	6	1	12	12	7.5	24.4×
		0.01%	-0.004	2.8	6	1	18	6	7.4	25.0×
		0.1%	0.004	1.9	8	1	19	3	7.7	23.9×
		1%	0.111	2.6	9	0	15	7	7.7	23.8×
		2%	0.031	3.2	6	1	11	13	7.8	23.6×
		5%	0.036	3.0	6	2	8	15	8.0	23.0×

Table 7.5: Performance of the coarse-to-fine heuristic.

Config.	Set	Req. ΔQ	Avg. Δq	Avg. Δp	Result classification =	+1	≥ 2	≥ 5	Avg. calls	speed-up
Default	good	0%	-0.162	-0.3	22	0	6	3	13.2	14.0×
		0.01%	-0.154	-0.2	22	0	7	2	12.5	14.7×
		0.1%	-0.134	0.1	24	0	7	0	8.4	21.9×
		1%	0.049	0.1	26	5	0	0	2.0	90.5×
		2%	0.208	0.3	23	6	2	0	1.4	132.7×
		5%	0.541	0.2	24	6	1	0	1.1	167.8×
Default	undecided	0%	-1.006	1.6	17	2	7	5	15.5	11.9×
		0.01%	-1.006	1.6	17	4	5	5	14.2	12.9×
		0.1%	-0.978	2.2	15	5	7	4	11.0	16.7×
		1%	-1.205	3.0	9	4	15	3	7.3	25.1×
		2%	-2.734	2.4	11	2	15	3	5.4	34.4×
		5%	-3.917	2.6	8	5	13	5	4.2	43.9×
Default	undecided+good	0%	0.000	0.0	31	0	0	0	21.6	8.5×
		0.01%	0.000	0.0	31	0	0	0	20.3	9.1×
		0.1%	0.000	0.0	31	0	0	0	13.5	13.6×
		1%	0.000	0.0	31	0	0	0	4.7	39.3×
		2%	0.000	0.0	31	0	0	0	3.7	49.6×
		5%	0.000	0.0	31	0	0	0	2.6	71.3×
Conservative	good	0%	-0.081	-0.1	29	0	2	0	22.9	8.0×
		0.01%	-0.081	-0.1	29	0	2	0	21.5	8.6×
		0.1%	-0.077	0.0	29	1	1	0	15.0	12.3×
		1%	-0.026	0.2	27	2	2	0	5.8	31.9×
		2%	0.013	0.3	26	3	2	0	4.4	41.9×
		5%	1.045	0.5	17	10	4	0	1.8	103.7×
Conservative	undecided	0%	0.000	0.0	31	0	0	0	24.0	7.7×
		0.01%	0.000	0.0	31	0	0	0	22.5	8.2×
		0.1%	0.000	0.0	31	0	0	0	15.6	11.8×
		1%	0.000	0.0	31	0	0	0	5.9	31.3×
		2%	-0.050	0.0	30	1	0	0	4.7	38.8×
		5%	0.176	0.2	23	6	2	0	3.6	50.9×
Conservative	undecided+good	0%	0.000	0.0	31	0	0	0	24.1	7.6×
		0.01%	0.000	0.0	31	0	0	0	22.7	8.1×
		0.1%	0.000	0.0	31	0	0	0	15.8	11.7×
		1%	0.000	0.0	31	0	0	0	6.0	30.7×
		2%	0.000	0.0	31	0	0	0	4.8	38.0×
		5%	0.000	0.0	31	0	0	0	3.4	54.8×

hits towards the largest mantissa or exponent. The second converges but ends with a too large representation. On average, if the mantissa is searched first, the process is likely to end up with a maximum of 23-bit selection for the mantissa which is over 15-bits more than the average optimal representation. Similar findings hold for the exponent, however, since the exponent representation saturates at 8-bit, on average over 3-bits more than optimal are returned. For the remaining cases, it is more beneficial to first search the mantissa and then the exponent representation rather than the other way around. The best working settings are (Mantissa, high) where on average 2.1-bits more than the optimal solution are spent to guarantee the quality constraint. The runtime is constant and leads to a speedup of about 25×.

Coarse versus fine search with adaptive evaluation

Table 7.5 shows the obtained results for the coarse-to-fine heuristic for the default and conservative set of parameters. The heuristic works reasonably well with default parameters and simply considering the good candidates for moderate quality constraints of $\geq 1\%$. In this case, the course step filters configurations such that in the remaining part only up to two configurations are considered for the final decision. Those results all exceed quality requirements and only cause cost overheads of up to 0.3-bits on average. For stricter quality requirements $< 1\%$ the heuristic might pre-select small sets of configurations that unfortunately do not contain a configuration that meets the quality constraint. In this case, the best result is returned which violates the quality constraint. Operating the algorithm in this domain is not recommended since the result is not guaranteed to contain a quality satisfying configuration, nor is the likelihood large enough that one might satisfy the user. Changing the filtering settings from default to conservative classifies more configurations into undecided candidates. This effect improves the results but slows the algorithm down. However, still, in 2 out of 31 cases quality constraints are violated. In contrast, operating the algorithm that all remaining undecided and good candidates are considered resolves all problems for the conservative and even for the default settings. However, that

way many additional configurations are required to be examined at full precision leading to a higher average evaluation cost.

7.5 Summary and conclusion

Table 7.6: Overview of properties and key results of transprecision search heuristics.

Heuristic	Quality constraint guarantee	Optimum guarantee	Constant runtime	Avg. bit-width increase	Speed-up
Exhaustive	YES	YES	YES	0-bit	ref.
Smart search	YES	YES	NO	0-bit	$> 5.3\times$
Parallel search[1]	YES	NO	YES	1.06-bit	$> 5.3\times$
Two-stage[2]	YES	NO	YES	2.1-bit	$> 24.6\times$
Coarse-to-fine[3]	NO	NO	NO	0-bit	$> 8.5\times$

[1] mantissa direction [2] mantissa direction with high pivot [3] default settings, with searching undecided and good classified configurations

In this chapter we demonstrated the following: We identified the importance of heuristics to quickly find good and optimal transprecision configurations faster. We studied three heuristics that reduce the number of required evaluations and one heuristic, the coarse-to-fine heuristic, that reduced the effort during evaluation of the quality. We evaluated the four heuristics on a representative set of tuning tasks of deep learning models where the optimal solution is known as summarized in Table 7.6. All of the considered heuristics have operating points where they outperform the others in at least one aspect. Relaxing required properties leads to much faster execution times. The coarse-to-fine search is an interesting heuristic that does not give any guarantees but still achieves optimal results in all of the considered cases at a decent speed.

Chapter 8

Optimization for IoT devices with given constraints

Many industrial workflows rely on automation and optimization. The higher the degree of automation, the simpler the elaboration and design of even more complex solutions. Fully automated pipelines that accept customized configurations easily express an optimization problem, where even-better solutions are found by using better configurations. If such configuration spaces are small enough, exhaustive evaluations directly lead to the global optimal solution. However, in relevant cases, configurations spaces might be large. They scale exponentially in the number of tunable hyper-parameters. Such optimizations turn out to be either computational infeasible, or at least questionable if the obtained improvements would be worth the invested computing costs. In such settings, by considering only the feedback of a small set of considered hyper-parameters, search heuristics find good enough configurations in affordable time.

The term *auto machine learning*, in short *autoML*, refers to situations where automation and optimizations are applied to an end-to-end machine learning workflows. Parameters of different stages are considered to optimize the solution. That includes settings from

data augmentation, model definition, and the learning algorithm itself, as outlined in Section 1.1. *AutoML* enhances solutions at the cost of extra computing time and succeeds when working with new data where little insights are known a priori.

Designing an economically viable artificial intelligence system has become a formidable challenge in view of the increasing number of published methods, data, models, newly available deep-learning frameworks as well as the hype surrounding special-purpose hardware accelerators as they become commercially available. The availability of large-scale datasets with known ground truths [22, 59, 132, 167–175] and the widespread commercial availability of computational performance—usually achieved with graphic-processing units (GPUs)—has driven the current growth of and strong interest in deep learning and the emergence of related new businesses. Smart homes [176], smart grids [177] and smart cities [178] trigger a natural demand for the Internet of Things (IoT), which are products designed to be low in cost and feature low energy consumption and fast reaction times due to the inherent constraints given by final applications that typically demand autonomy with long battery lifetimes or fast real-time operation. Experts estimate that there will be some 30 billion IoT devices in use by 2020 [179], many of which serve applications that benefit from artificial intelligence deployment.

In this context, we propose an automatic way to design deep-learning models that satisfy user-defined constraints specifically tailored to match typical IoT requirements, such as inference latency bounds. Additionally, our approach is designed in a modular manner that allows future adaptations and specialization for novel network topology extensions to different IoT devices and lower precision contexts.

This chapter is organized as follows. Section 8.1 describes related work, Section 8.1 introduces the core design procedures, Section 8.2.4 details and merges a full synthesis workflow, Section 8.3 presents and discusses the obtained results, and Section 8.4 concludes our findings.

8.1 Related work for network architecture search

Automated architecture search has the potential to discover better models [29–35, 40]. However, traditional approaches require a vast amount of computing resources or cause excessive execution times due to the full training of candidate networks [36]. Early stopping based on learning-curve predictors [37] or transferring learned wights shortens run-times [38]. A method called train-less accuracy predictor for architecture search (TAPAS) demonstrates how to generalize architecture search results to new data without having to train during the search process [39]. Architecture searches face the common challenge of defining the search space. Historically, new networks were developed independently by expert knowledge that outperforms previously found networks generated by architectural searches. In such cases, very expensive reconsiderations led to follow-up work to account correctly for a richer search space [180, 181]. Recent progress in the field, such as MnasNet [41] and FBNet [42], tailor the search by optimizing a multi-objective function including inference time on smartphones. MnasNet trains a controller that adjusts to more optimal sample models in terms of multi-objectivity. FBNet trains a supernet by a differentiable neural architecture search (DNAS) in a single step and claims to be 420× faster by avoiding additional model training steps. In contrast to solving a joint optimization problem in one step, our proposed union of narrow-space searches takes a modular approach that separates the search process of finding architectures that strictly satisfy constraints from the training of candidate networks. That way, we can analyze 10,000 architectures with no training cost and select only a small subset of suitable candidates for training.

Compression, quantization and pruning techniques reduce heavy computational needs based on the inherent error resilience of deep neural networks [182]. Mobile nets [28] or low-rank expansions [183] change the topology into layers that require fewer weights and reduce workloads. Quantization studies the effect of using reduced precision floating-point or fixed-point formats [166, 184], whereas compression attempts to reduce the binary footprint of activation and weight maps [185]. Pruning approaches avoid computation by enforcing sparsity

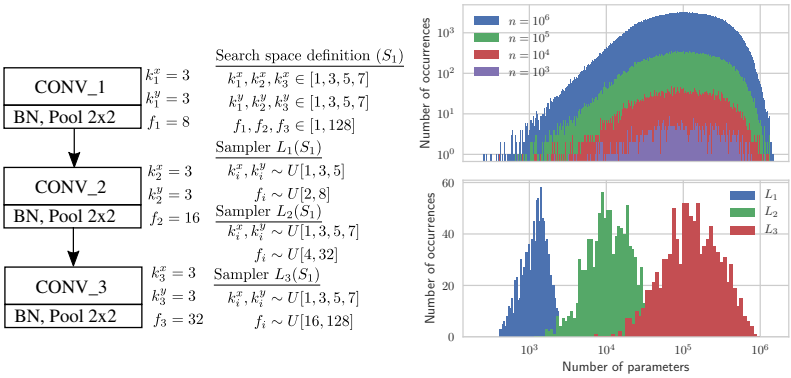


Figure 8.1: Left: Three-layer architecture. Middle: Default configuration of search space with restricted sampling laws. Right: Statistics of number of parameters obtained by sampling up to one million networks from the base configuration space and 1000 networks from the restricted sampling laws.

[186]. We extensively use the developed integration of floatx into PyTorch as detailed in Section 6, to assess data format-specific aspects of networks. The novelty of our work is that we jointly evaluate network topologies in combination with reduced precision.

8.2 Narrow-space architecture search

It is challenging to define a space S that produces enough variation and simultaneously reduces the probability of sampling suboptimal networks. We propose narrow-space architecture searches, where results are obtained by aggregating n independent searches $S = \bigcup_{i=1}^{n} S_i$. As a good search space should satisfy $S_r \subset S$, where $S_r = \{M_1, ..., M_n\}$ is a set of reference models, we construct S by designing narrow spaces that obey $M_i \in S_i$ in order to guarantee $S_r \subset S$. Instead of considering one large space, we have specialized search spaces that produce simple sequence structures with residual bypass operations (ResNets [24]) to even high fan-out and convergent structures such as they occur

Table 8.1: Search spaces induced from established reference models

Space	Reference model	Params	Ops	Accuracy	
				Ours[1]	Ref[2]
S_1	DenseNet121 [27]	7.0M	898.1M	94.13%	95.04%
S_2	MobileNetV2 [189]	2.3M	94.6M	92.94%	94.43%
S_3	GoogLeNet [25]	6.2M	1.5G	93.55%	-
S_4	PNASNetA [190]	135.5K	29.2M	83.85%	-
S_5	ResNeXt29_32x4d [191]	4.8M	779.6M	93.46%	94.73%

[1] reproduced results with our training limited to 100 epochs
[2] reference results of third-party implementation [192] with high-effort training of 350 epochs

in the Inception module [187] or DenseNets [188]. Aggregation allows results to be extended easily with a tailored narrow-space search for new reference architectures. Next, we define a set of distribution law configurations $L_1(S_i), ..., L_k(S_i)$ that allow samples to be drawn in a biased way such that models satisfy the properties of interest. Figure 8.1 illustrates the advantages over a uniform distribution among valid networks. Consider a space of three-layer networks with allowed variations in kernel shapes in $\{1, 3, 5, 7\}$ and output channels in $[1, 128]$ leading to $|S| = 4^6 * 128^3 = 8.6 * 10^9$ network configurations.

Figure 8.1 shows the statistics for up to 10^6 samples compared with sampling only 1000 samples using restricted samplers L_1, L_2 and L_3. Restricted random laws efficiently generate networks of interest, in contrast to a uniform sampler that fails to deliver high sampling densities in certain regions. For example, only 132 out of 10^6 networks have fewer than 1000 parameters.

8.2.1 Narrow-space and sampling law definition

We define each narrow-space architecture search and its sampling laws according to the following design goals: First, only valid models are generated with a topology that resembles and includes the original model. Second, the main model-specific parameters are varied, and efficient models are obtained mainly by lowering channel widths in

Table 8.2: Architecture search space definition S_1 with different sampling laws for DenseNets.

Law	Cardinality	DenseNets parameters		
		$n_i, i \in \{1,2,3,4\}$	g^*	r^{**}
L_0	$3.3 \cdot 10^9$	$[1, 32]$	$[1 - 32]$	$[0.0, 1.0]$
L_1	$2.0 \cdot 10^6$	$[1, 8]$	$[1 - 8]$	$[0.2, 0.8]$
L_2	$6.3 \cdot 10^7$	$[1, 16]$	$[1 - 16]$	$[0.2, 0.8]$
L_3	$1.0 \cdot 10^9$	$[1, 32]$	$[1 - 32]$	$[0.5, 0.8]$

* g is the growth rate, ** r is the reduction rate

convolutional layers and reducing the number of topological replications. Third, all random laws are defined following a uniform distribution over available options, where the lower and upper limits were used as a way to bias the models to span several orders of magnitude targeting the range of parameters and flop counts relevant for IoT applications.

Table 8.1 lists the five search spaces used in this work that are based on established models. Typical models consist of 2M up to 7M of parameters and cause workloads from 94.6 million up to 1.5 billion floating-point operationss (FLOPs) and are too large for fast implementations on a targeted IoT device. DenseNets [27] exists in common variants, 121, 161, 169, and 201 and we used the smallest variant (DenseNet121) as starting point. We reproduced the accuracy for all architectures by running our training procedure as detailed in Section 8.2.6 where we used an upper limit of 100 epochs and compare it with the claimed reference accuracy from the source from where we obtained the architecture implementation in PyTorch [62]. The latter values are slightly higher but they are obtained with a high effort training that runs for a fixed amount of 350 epochs. Additionally, the later source does not state the mean and variance of the training process neither is it completely clear if the values are obtained in a one-shot training or if the best values have been selected after repeating the training process several times. In contrast, we decided to follow a pragmatic but efficient approach of evaluating each architecture

Table 8.3: Architecture search space definition S_2 with different sampling laws for MobileNets.

Law	Cardinality	MobileNets parameters, $i \in [1, 7]$					
		f_{in}	e_i	f_i	n_i	s_i	f_{out}
L_0	$5.2 \cdot 10^{34}$	$[1, 128]$	$[1, 8]$	$[1, 256]$	$[1, 4]$	$[1, 2]$	$[1, 1280]$
L_1	$3.6 \cdot 10^{33}$	$[16, 128]$	$[1, 6]$	$[16, 256]$	$[1, 4]$	$[1, 2]$	$[128, 1280]$
L_2	$2.0 \cdot 10^{28}$	$[16, 64]$	$[1, 4]$	$[16, 128]$	$[1, 3]$	$[1, 2]$	$[128, 512]$
L_3	$3.1 \cdot 10^{21}$	$[16, 32]$	$[1, 2]$	$[16, 64]$	$[1, 2]$	$[1, 2]$	$[128, 256]$
L_4	$4.0 \cdot 10^{19}$	$[16, 32]$	$[1, 2]$	$[4, 32]$	$[1, 2]$	$[1, 2]$	$[64, 128]$
L_5	$1.4 \cdot 10^{15}$	$[16, 32]$	$[1, 2]$	$[2, 8]$	$[1, 2]$	$[1, 2]$	$[16, 64]$
L_6	$4.3 \cdot 10^{13}$	$[4, 8]$	$[1, 2]$	$[2, 8]$	$[1, 2]$	$[1, 2]$	$[12, 16]$

only once and to limit training effort to an affordable value of 100 epochs. This decision is motivated by the fact that we want the same training procedure to be applied to over 3,000 models. Training evaluations with high-effort would cause 3.5× more computational costs and repeating experiments to deliver statistics would at least require a repetition factor of 5×. Both aspects together cause a 17.5× increase in computation cost. In our opinion, if we are willing to pay such an increase, it would be more interesting to use an affordable approach and invest the additional budget into investigating more architectures. The increased effort would allow investigating over 50,000 network architectures. Next, we define the sampling laws and parameters used to manually enforce smaller variants of networks within the defined spaces.

DenseNets [27] consists of four stages, each repeating DenseNet unique blocks. We identified the stage-specific number of repetitions, the growth rate and the reduction factor as relevant hyper-parameters that we modify. Table 8.2 specifies the sampling laws. In this case, we decided to clip the repetition factor at 32 which additionally includes the configuration of DenseNet161. The normalized reduction factor is sampled with a step size of 0.01.

Table 8.4: Architecture search space definition S_3 with different sampling laws for GoogLeNets.

Law	Cardinality	GoogLeNets parameters, $i \in [1, 9]$						
		f_0^0	f_i^1	f_i^2	f_i^3	f_i^4	f_i^5	f_i^6
L_0	$3.4 \cdot 10^{122}$	$[16, 256]$	$[16, 384]$	$[16, 192]$	$[16, 384]$	$[16, 48]$	$[16, 128]$	$[16, 128]$
L_1	$2.8 \cdot 10^{119}$	$[16, 256]$	$[16, 384]$	$[16, 192]$	$[16, 384]$	$[16, 48]$	$[16, 128]$	$[16, 128]$
L_2	$1.1 \cdot 10^{97}$	$[16, 256]$	$[16, 64]$	$[16, 64]$	$[16, 64]$	$[16, 32]$	$[16, 32]$	$[16, 32]$
L_3	$9.8 \cdot 10^{51}$	$[16, 256]$	$[16, 32]$	$[8, 16]$	$[16, 32]$	$[4, 8]$	$[4, 8]$	$[4, 8]$
L_4	$6.3 \cdot 10^{39}$	$[16, 128]$	$[4, 8]$	$[4, 8]$	$[4, 8]$	$[4, 8]$	$[4, 8]$	$[4, 8]$
L_5	$5.2 \cdot 10^{26}$	$[8, 16]$	$[4, 6]$	$[4, 6]$	$[4, 6]$	$[4, 6]$	$[4, 6]$	$[4, 6]$
L_6	$5.0 \cdot 10^{38}$	$[8, 16]$	$[2, 6]$	$[2, 6]$	$[2, 6]$	$[2, 6]$	$[2, 6]$	$[2, 6]$

$f_{i+1}^0 := f_i^1 + f_i^3 + f_i^5 + f_i^6$ for $i \geq 0$, recursive definition such that next input shape matches the previous output shape

Table 8.5: Architecture search space definition S_4 with different sampling laws for PNASNet-A.

Law	Cardinality	PNASNet-A parameters, $i \in \{1, 2, 3\}$			
		n_i	f_1	d_2	d_3
L_0	$3.5 \cdot 10^6$	$[1, 12]$	$[1, 128]$	$[1, 4]$	$[1, 4]$
L_1	$3.1 \cdot 10^6$	$[1, 12]$	$[16, 128]$	$[1, 4]$	$[1, 4]$
L_2	$2.3 \cdot 10^5$	$[1, 8]$	$[16, 64]$	$[1, 3]$	$[1, 3]$
L_3	$9.7 \cdot 10^2$	$[1, 3]$	$[8, 16]$	$[1, 2]$	$[1, 2]$

Set $f_2 := f_1 \cdot d_2$ and $f_3 := f_2 \cdot d_3$.

MobileNetsV2 [189] consists of seven stages, each repeating MobileNetsV2 unique blocks. Each block is configured with four parameters, input number of channels, output number of channels, expansion factor, and a stride factor. To generate valid configurations, we define the first input channel number separately, since the subsequent input shape follows directly from the previous block configuration. Additionally, we restrict the stride factor per stage to either be one or two and we further require to sample exactly three twos and four ones. The reason for this choice is due to the stride parameter directly influences the spatial shape of tensors and the limitation ensures fixed downsampling over three steps from 32×32 to 4×4. Additionally, the existing intermediate last convolutional layer is separately parametrized and is used as in the original reference model as a transition layer between the last block and the final linear classifier. Table 8.3 states the sampling law definitions.

GoogLeNet [25] is composed of the characteristic Inception module, which is defined through seven intermediate channel depths. The full network is grouped into three stages, first a convolutional prelayer, second, and third a max-pooling separates sequences that are built from two, five, and two Inception modules. Table 8.4 defines the sampling laws. We choose parameter specific upper bounds oriented on the reference implementation.

PNASNet-A [190] consists of three stages that are build by repeating cell-A type of blocks. The stages are separated by downsampling

Table 8.6: Architecture search space definition S_5 with different sampling laws for ResNeXt.

Law	Cardinality	ResNeXt parameters, $i \in [1,3]$		
		n_i	f_i	c_i
L_0	$9.5 \cdot 10^{14}$	$[1,3]$	$[1,64]$	$[1,512]$
L_1	$2.1 \cdot 10^{10}$	$[1,3]$	$[4,64]$	$[1,512/f_i]$
L_2	$2.4 \cdot 10^{8}$	$[1,3]$	$[4,32]$	$[1,128/f_i]$
L_3	$1.5 \cdot 10^{5}$	$[1,2]$	$[4,8]$	$[1,32/f_i]$

layers that are implemented as cell instances with a stride of two. Table 8.5 defines the sampling laws that affect the number of block repetitions and the number of channels used in the block, where f_2 and f_3 are relatively defined to the output shape of previous stages.

ResNeXt [191] is the improvement over the typical ResNet [24] structure. It consists of a three-stage architecture where each stage repeats the bottleneck block n_i times, for $i \in \{1,2,3\}$. The block consists of the typical residual connection and follows a bottleneck design where grouped convolutions are used to reduce the kernel size. We define in the search space with the bottleneck base width f_i and the cardinality c_i. However, the total channel size that is invoked during the grouped convolution operation is of width $f_i \cdot c_i$. Since we want to limit the product $f_i \cdot c_i$ but we also require it to be devisable by either f_i and c_i we decided to randomly sample the later and restrict the cardinality upper bound to be l_{high}/f_i, where l_{high} denotes the upper product limit and f_i is dependent on the current sampling of the base depth. Table 8.6 summaries the defined random laws.

8.2.2 Precision analysis

Precision analysis evaluates model accuracies for models having reduced precision representations. Following general methodology, we perform precision analyses on the backend device that has different execution capabilities than current or future targeted IoT devices. This methodology enforces emulated computation throughout the analysis to assess accuracy independent of the target hardware. Low precision

can be applied to model parameters, to the computations performed by the models and to the activation maps that are passed between operators. Here we follow the extrinsic quantization approach [166], where we enforce a precision caused by the reduced type $T_{w,t}$ of storage width $1+w+t$ to be applied to all model parameters and all activation maps that are passed between operations. Our analysis follows the IEEE 754 standard [116], which defines storage encoding, special cases (NaN, Inf), and rounding behavior of floating-point data. A sign s, an exponent e and the significand m represent a number $v = (-1)^s \cdot 2^e \cdot m$, where the exponent field width w and the trailing significant field width t limit dynamic range and precision. Types $T_{5,10}$ and $T_{8,23}$ correspond to standard formats *half* and *float*. Our experiments are based on a PyTorch [62] integration of the GPU quantization kernel based on the high-performance floatx library [193], which implements the type $T_{w,t}$. The development effort of Section 6 enables a fast precision analysis that allows us to evaluate more than 3,000 models with a full grid search of 184 types ($w \in [1, 8], t \in [1, 23]$) of the entire validation data.

8.2.3 Performance characterization on hardware

To evaluate model execution performance on the IoT target device, we perform a calibration to assess the execution speed of the models of interest. Despite many choices of deep-learning frameworks, ways of optimizing code depending on compilation or software version and even several hardware platforms that accelerate deep learning models, we formulate the performance characterization in a general manner and as decoupled as possible from the topology architecture search and the precision analysis to facilitate subsequent extensions. Performance measurements on the IoT device are affected by explicit and implicit settings. We demonstrate our search algorithm with performance measurements featuring the fewest assumptions and requirements regarding runtime. To that end, we selected Raspberry-Pi 3(B+) as a representative low-cost IoT device. It features a Broadcom BCM2837B0, quad-core ARMv8 Cortex-A53 running at 1.4 GHz. The board is equipped with 1 GB LPDDR2 memory [194]. The Raspberry-Pi 3(B+) belongs to the general-purpose device category that is shipped with peripherals (WiFi, local area network (LAN),

Bluetooth, and universal serial bus (USB), high definition multimedia interface (HDMI)) and a full operating system (Raspbian, a Linux distribution). It is available for about $35 [195].

In the following, we justify the choice of using the Raspberry-Pi 3(B+) as representative IoT device. First, it should be mentioned that there is a current emerging trend in industry and research that pushes to improve hardware for artificial intelligence (AI) by either improving performance, reducing power consumption or providing better trade-offs in terms of power/performance ratios or hardware cost versus the on-device supported features. In terms of thinking through IoT driven business cases, the fact that new hardware appears requires to benchmark and rethink on what HW product a certain IoT application should be built. We are aware of the existence of tens of ASIC or FPGA solutions that might be selected for business legitimated reasons as a target edge inference system.

We think that the crucial factors for a successful IoT deployment strategy cover the following points with an importance that is application specific:

- reliability;
- user/developer friendly software ecosystem;
- modular integration or extensions of different functionality;
- typical IoT support;
- cost efficient system.

We decided that in this work we focus on the algorithm. However, since we are aware of many choices and good reasons for a certain HW solution, we developed our approach such that the main functionality is decoupled from the actual HW implementation. Especially, populating a database with the current results that are obtained with expensive training for obtaining the model accuracy, can easily be reused later on for any hardware platform by just implementing the inference and timing measurement setup in order to obtain the new HW calibration information. In this work, we focus on the most general use-case that causes the least amount of requirements for the underlying hardware. That way, we identified the Raspberry-Pi

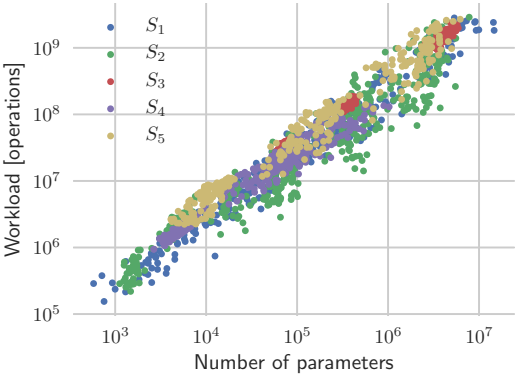


Figure 8.2: High correlations between the size of networks and the computational workload required for a single inference. Both metrics are analytical properties of the network architectures and are known at design and construction time of the model.

3(B+) as general purpose quad-core architecture as a suitable IoT device candidate. The Raspberry-Pi proves its marketability by the fact that it has been shipped over 25 million times by February 2019 [196]. Even though there are competing products that are specially tailored for AI deployment, the choice of selecting a general-purpose platform equipped with a Linux operating system comes with obvious advantages, such that it enables to reuse established software and solutions can be easily extended to any needs. In contrast to dedicated AI accelerators that are shipped as USB dongles, potentially required features such as Ethernet, WiFi, SDCard slot, or USB ports are already included in the Raspberry-Pi 3(B+). Even though we are aware that a general purpose architecture cannot compete in some performance metrics with a dedicated AI product, we argue that our work is especially insightful since we cover the more challenging case on optimizing for a performance limited device. In our view, it is plausible enough to argue that a more performant device will automatically deliver better results. We aim to support various HW platforms with different deployment flows in future work.

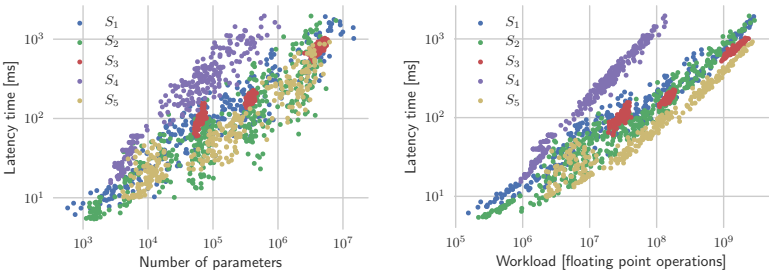


Figure 8.3: Runtime-dependent latency measured as function of network size (left) and network workload (right). The latency is best correlated with the workload when accounting for different search space-specific characteristics.

Next, we describe the deployment flow for the Raspberry-Pi 3(B+). Even though our back-end algorithms, as well as our training routine, is implemented in PyTorch, we still aim to remove the back-end dependency in order to be open and to ease later migration to new target platforms, frameworks, and ecosystems. To that end, we decided to export all models according to the open neural network exchange (ONNX) format [197]. We decided to use caffe2 as target device runtime for the exported ONNX models. We build the caffe2 framework directly from a full source compilation with all default parameters on the Raspberry-Pi 3(B+) and we ensured that the produced code is using the ARMs NEON library [198] for fast computation. We wrote a light script to import the produce ONNX models and we trigger a sequence of inferences for a single image. In our work, all timing results have been obtained by averaging wall clock times over ten repetitions. We used a batch size of one to minimize latency and internal memory requirements. The latency study covers many relevant use cases, for example the classification of sporadically arriving data within a short time to prolong battery lifetime or frame processing a video stream, where the classification must be completed before the next frame arrives.

For each model, we consider two analytical properties, the number of trainable parameters and the workload measured as the number of

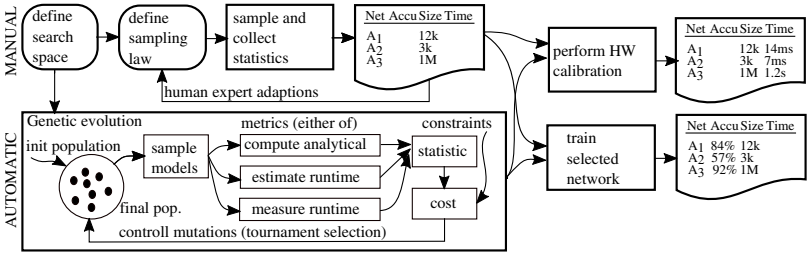


Figure 8.4: Manual and automatic workflow. First, sampling laws are defined to generate models of interest. Second, models are calibrated to check latency on the IoT device, even if they are not yet trained. Third, models are trained to achieve accuracy. As training is the most expensive task, it is essential to limit the number of trained models to candidates of interest only.

floating-point operations required for inference. The calibration step relates analytical properties with execution performance and allows us to separate runtime metrics. Figure 8.2 shows high correlations between the number of parameters and the workload. Figure 8.3 shows that using either of the analytical property of the model allows to predict the latency on the Raspberry-Pi 3(B+) device. Workload and parameters follow a similar scaling over five orders of magnitude with homogeneous variations. The dynamic range of the latency spans more than two orders of magnitude with higher variations for larger models. However, owing to the compute-bound nature of the kernels, the workload is a better indicator of latency time than the number of parameters.

8.2.4 Fast cognitive design algorithms

In this section, we leverage the architecture search, the precision analysis, and the hardware calibration steps to synthesize case-specific solutions that satisfy given constraints. We address two tasks: First, the constraint search solves for the model that best satisfies given constraints. Second, the Pareto front elaboration provides insights

into tradeoffs over the entire solution space. The two tasks are related. Solving the first task on a grid of constraints provides solutions to the second task, whereas filtering the latter based on the given constraints yields the former. Both tasks are solved by manually and automatically by defining the sampling law configurations on the same set of narrow-search spaces as shown in Figure 8.4. In the manual task, collected statistics of analytical network properties provide quick feedback to adapt the settings to cover the range of interest. For a fair comparison of the manual and automatic workflows, we assume throughout our experiments that the expert has no further feedback knowledge about model accuracy. Additionally, network runtime performance metrics can be measured on the target device or estimated from calibration measurements. Next, depending on the task type, either a few candidate networks that satisfy constraints or a full wave of networks are selected for training. Large-scale training takes the most time—as each training job is of complexity $O(n_{train}C_{model}E)$—proportional to the amount of training data, model complexity and the number of epochs for which the model is trained.

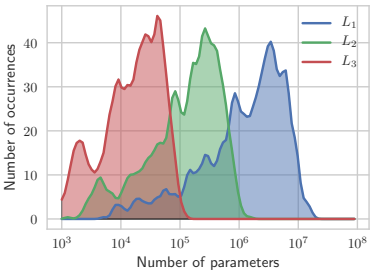
We designed a genetic and clustering-based algorithm to automatize the design of sampling laws. We define the valid space with a list of variables with absolute minimal and maximal ratings. A sampling law $L(S_i)$ is defined as an ordered set of uniform sampling laws $L = (U_x[l_x, h_x], ...)$ with lower and upper limits l_x and h_x per variable x. The genetic algorithm automatically learns the search space-specific sampling law limits $[l_x, h_x]$. The cost function is defined in a two-step approach. First, the statistic $(\mu_m, \sigma_m) := E_m^n(L)$ is estimated by computing means and standard deviations over the metric m extracted from the n generated topologies. Second, cost is computed as $c((\mu_m, \sigma_m), (\tau_1, \tau_2)) := |\mu_m - \sigma_m - \tau_1| + |\mu_m + \sigma_m - \tau_2|$ in order that the high density range of the estimated distribution coincides with a given interval (τ_1, τ_2). We avoided definitions based on single-sided constraints such as $\mu < \tau$ because such formulations might be satisfied trivially (using the smallest network) or by undesirable laws having wide or narrow variations. We used the tournament selection variant of genetic algorithms [199] and defined mutations by randomly adapting the sampling law of hyper-parameters l_x and h_x. We used an initial population of $n_{init} = 100$ and ran the algorithm for $n_{steps} = 900$ steps while using $n_{eval} = 10$ samples to estimate

mean and standard deviations per configuration. This way, one search considers $(n_{init} + n_{steps}) * n_{eval} = 10,000$ networks. As the final population might contain different sampling laws of similar quality, we performed spectral clustering [200] to find $k = 10$ clusters with similar sampling laws. We assembled a list of the most different top-k laws by taking the best-fit law per cluster.

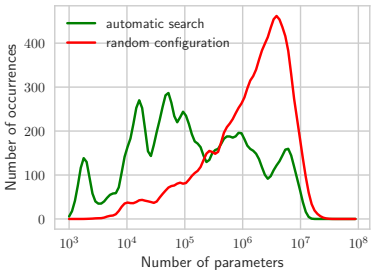
To elaborate the entire search space with a Pareto optimal front, we split each decade into three intervals $[\tau, 2\tau, 5\tau, 10\tau]$ and define a grid for $\tau = 10^3, 10^4, 10^5, 10^6$ spanning five orders of magnitude. We ran the genetic search algorithm several times by setting the target bounds (τ_1, τ_2) in a sliding-window manner over consecutive values from the defined grid. Finally, we accumulated results from twelve genetic searches, each of which found ten sampling laws, where we sampled each law $n_{val} = 100$ times to obtain the statistic of 12,000 network architectures per narrow-space search.

8.2.5 Statistical properties of generated networks

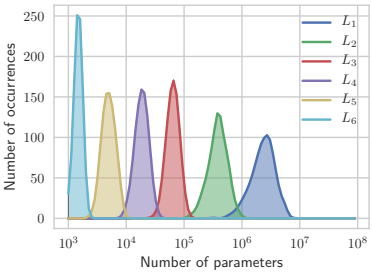
For the defined search spaces and sampling laws, we collected statistics over 1,000 networks that are presented in Figure 8.6. We targeted to cover the full domain in $[10^3, 10^7]$. Some search spaces, such as S_1, S_4, and S_5 are quickly covered with three simple configurations. Other search spaces, such as S_2 and S_3, lead to narrower distributions where we decided to add three additional sampling laws to cover the lower domain. Even though the manual search covers the region of interest nicely, human expertise is required to define the parameters of the laws L_1 to L_6 correctly. The right-hand side of Figure 8.6 shows statistics obtained when networks are obtained by uniformly sampling each parameter in its full domain (according to the law L_0) and when they are obtained with automatically generated sampling laws that are adjusted with our proposed genetic algorithm. The naive sampling approach in the entire search space produces a narrow distribution and is strongly skewed towards larger networks. The base law of the original definition has a high impact on where the actual mass of the distribution concentrates. The mass densities are around 10^6 parameters for S_1 and S_2. S_3 and S_4 have the center of mass above 10^7 parameters which cause difficulties for the genetic algorithm to converge towards the low end of the domain. In contrast,



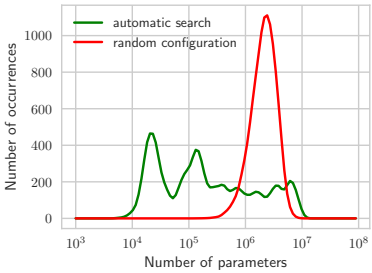
(a) Manual laws on S_1



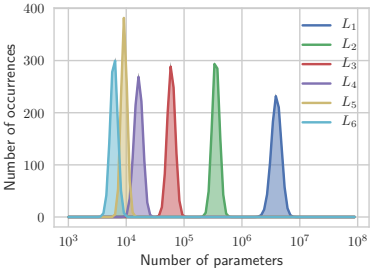
(b) Automatic law on S_1



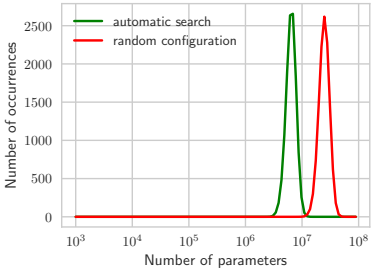
(c) Manual laws on S_2



(d) Automatic law on S_2

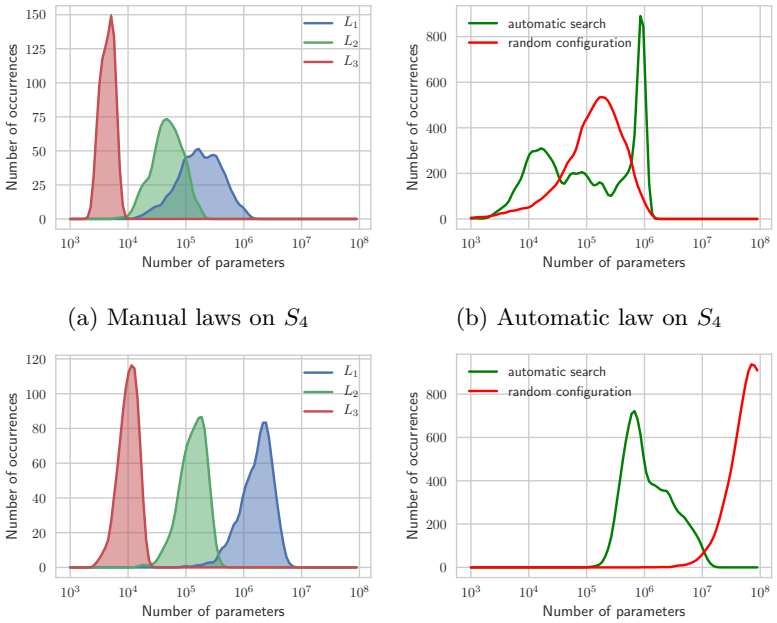


(e) Manual laws on S_3



(f) Automatic law on S_3

the genetic algorithm equalizes the distribution and provides samples that cover much higher dynamic ranges, extending the scale especially for smaller networks without manually restricting the architecture.



(a) Manual laws on S_4

(b) Automatic law on S_4

(c) Manual laws on S_5

(d) Automatic law on S_5

Figure 8.6: Statistic over manual (left) and automatic generated (right) networks for all search spaces S_1 up to S_5. By manually designing the sampling law, a human expert can reasonably adjust and focus the distribution into regions of interest, either close to a target constraint and in a general way to cover five orders of magnitude.

8.2.6 Training setup

We conducted all training experiments in a controlled environment where we trained from scratch for each candidate architecture. We used PyTorch version 0.4.1 as development framework and run on IBM Power8 or Power9 nodes equipped with either P100 or V100 GPUs. We used standard on-the-fly data augmentation during training that pads images with 4 pixels and randomly crops the image to 32×32 pixels, apply horizontal flipping with a probability of 0.5 and

finally normalizes pixel values to zero mean and unit variance. During testing, the original 32×32 images are directly normalized and feed into the models. For training, we used stochastic gradient descent with a batch size of 128 samples configured with an initial learning rate of 0.01, a momentum of 0.9, and a weight decay factor of $5 \cdot 10^{-4}$. We used a fixed scheduling schema where the learning rate is divided by a factor of 10 at epoch 40 and 70 and we limit training to stop at 100 epochs.

8.3 Results

To study our algorithm, we ran full design-space explorations on the well-established CIFAR10 [22] classification task and compared our results with those obtained with established reference models. Figure 8.7 shows the tradeoff between model size and accuracy, including manually and automatically generated results of the aggregate search spaces. The Pareto optimal front follows a smooth curve that saturates towards the best accuracy obtainable for large models. The number of parameters is logarithmic and the accuracy scales linearly. Even very small models with fewer than 1000 parameters can achieve accuracies of greater than 45%. The accuracy increase per decade of added parameters is on the order of 30%, 15%, 3% and $< 2\%$ points and then decreases very quickly. This effect allows us to construct models having several orders of magnitude fewer parameters. It also provides economically interesting solutions for IoT devices that are powerful enough to process data in real time. We compare our results with three sources of reference models: (a) traditional reference models, (b) ProbeNets [201] that are designed to be small and fast and (c) models designed to run on the PULP platform [202]. Traditional models include 30 reference topologies including variants of VGG [23], ResNets [203], GoogleNet [25], MobileNets [189] dual-path nets (DPNs) [26] and DenseNets [27], where most of them (28/30) exceed 1 M parameters. ProbeNets were originally introduced to characterize the classification difficulty and are considerably smaller by design [201]. They act as reference points for manually designed networks that cover the relevant lower tail in terms of parameters. In the

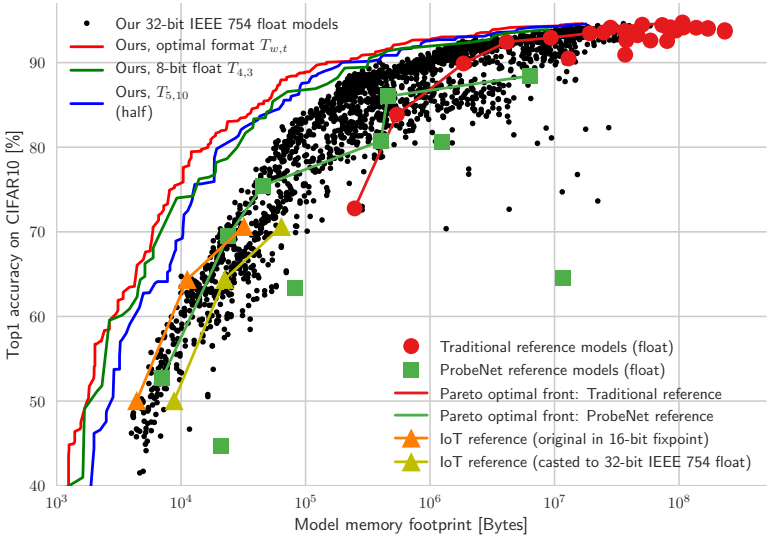


Figure 8.7: Results of our architecture search compared with reference models. Each dot represents a model according to its size and the obtained accuracy on the CIFAR10 validation set. Our search finds results over five orders of magnitude and, in particular, finds various models that are much smaller than out-of-the box models. In the restricted IoT domain, our search delivers models that outperform the reference with a wide margin for fixed constraints.

IoT-relevant domain (<10 M parameters), our search outperforms all the listed reference models.

The top three fronts in Figure 8.7 show the results of our precision analysis. For each trained model, we evaluated the effect of running models with all configurations of type $T_{w,t}$ and plot the Pareto-optimal front. We considered three cases: (1) running all models with half-precision, (2) running all models with the type T_{4_3}, which is the best choice for types that are 8-bits long, and (3) running each model with its individual best tradeoff type $T_{w,t}$. We demonstrate empirically that reduced precision pushes the Pareto optimal front. Under a given

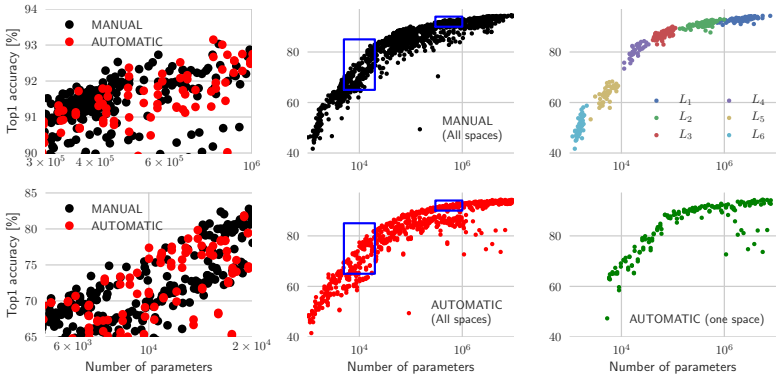


Figure 8.8: Left: Zoomed view of direct comparison; manual and automatic searches perform equally well. Middle: Manual and automatic search results. In the manual case, clusters are visible, whereas the automatic search sampled in a more homogeneous manner. Right: Results for one narrow-space search with marked clusters matching Figure 8.6.

memory constraint, accuracy improves by more than 7% points for half and by another 1% points or more for the model individual format.

Figure 8.8 shows details of manual and automatic searches, both of which yield very similar results. The right-hand graphs show results obtained for one narrow-space search, where manually defined sampling laws led to clusters. The automatic search covered a similar range homogeneously. Figure 8.9 shows inference times when the same set of models is executed on a Raspberry Pi 3(B+). Similarly, providing additional latency time for small models results in dominant accuracy gains, however, large models only slightly improve accuracy even when using more complex models that require long evaluation times.

Figure 8.10 demonstrates the scalability of our approach. We applied our search for three constraints $\tau = 10^3, 10^4, 10^5$ on thirteen datasets [201], where we spent a training effort of ten architectures

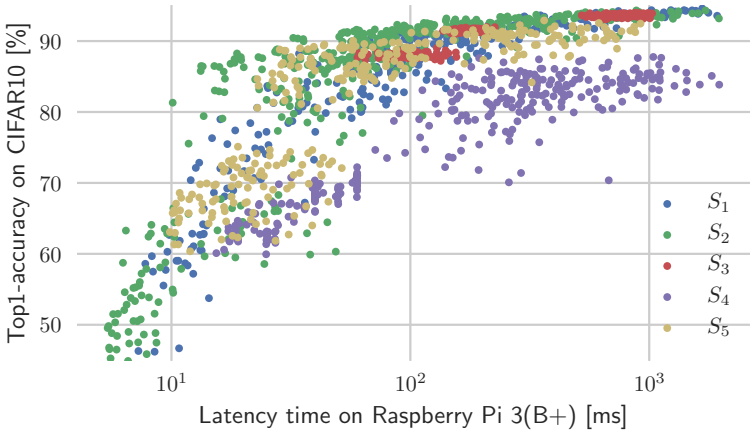


Figure 8.9: Final result showing the achievable tradeoffs between the IoT device measured model latency and model accuracy. Our search is able to deliver models that run below 10 ms on a Raspberry Pi 3(B+), which we took as a representative low-cost IoT device.

per dataset and constraint. The lines connect the best per constraint and dataset performing architectures.

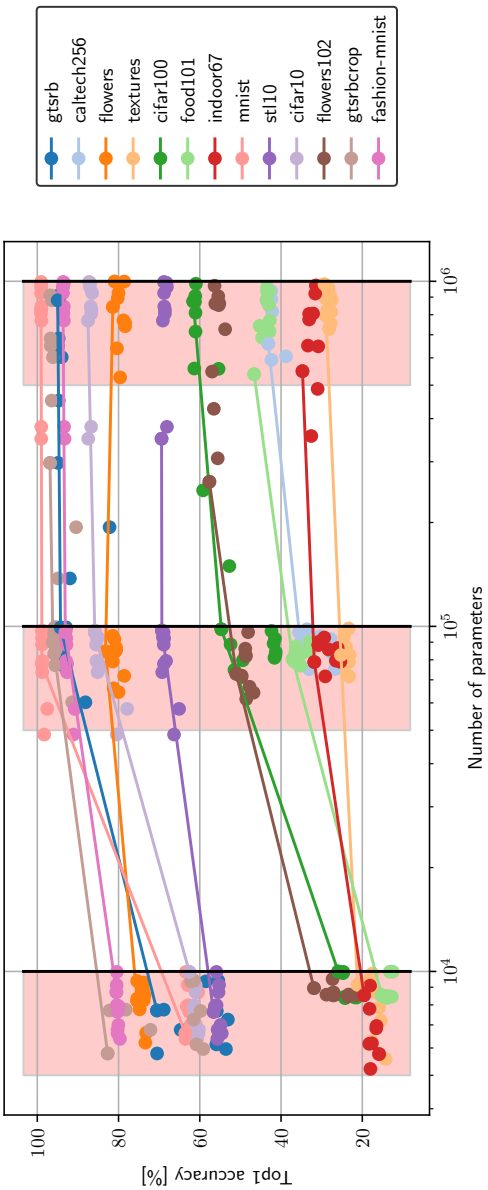


Figure 8.10: We demonstrate the scalability of our approach by applying our search to three constraints on thirteen datasets. Best models per dataset and constraint are connected with a line.

8.4 Summary and conclusion

In this chapter we studied the synthesis of deep neural networks that are eligible candidates to run efficiently on IoT devices. We propose a narrow-space search approach that leverages knowledge quickly from existing architectures and that is modular enough to be adapted to new design patterns. Manually and automatically designed sampling laws allow various models to be generated having sufficiently numerous parameters to cover multiple orders of magnitude. We demonstrate that reduced precision improves top-1 accuracy by over 8% points for constraint weight memory in the IoT-relevant domain. A strong correlation between model size and latency enables us to create small enough models that provide superior inference response latencies below 10 ms on an edge device costing only about $35.

Chapter 9

Conclusions

In this thesis, we have proposed the concept of transprecision computing as the new computing paradigm. Driven by Moors law, the computer industry is constantly following technology-based improvements. However, physical limitations, such as thermal dissipation, saturates economical viable operating frequencies and henceforth, the performance of computing systems. SIMD parallelism and multicore systems exploit parallelism to sustain the rising quest for better performance. In contrast to those approaches—that certainly remain and simultaneously evolve future computing systems—transprecision computing aims to reduce precision without quality degeneration to improve performance. We observed that approximate computing techniques followed the same spirit with great success in specific domains. Transprecision computing pursues the visionary goal of being applicable in general-purpose computing systems in various situations. System-level considerations include the simultaneous co-development of algorithms, software, and hardware. Performing these at once has been successfully demonstrated for specific tasks the justify ASIC development. However, those systems do not follow general concepts that apply to general-purpose computing. To that end, the developed transprecision computing concepts disentangle considerations in a modular way. First, target applications or systems are required

to be specified with expectations on quality and performance. Second, algorithm development and numeric evaluations lead to case-specific, domain-specific, or even general insights. To support adequate evaluations, we introduced a reduced precision library—floatx—in Chapter 5. Improved emulation speed and integrating floatx in PyTorch [62] opens the doors for large-scale elaborations. We address related problems, such as using heuristics to efficiently find good transprecision configurations in large search spaces. Additionally, often missing in related work, we assess the value of opportunity costs. To that end, transprecision computing needs to outperform alternatives. In Chapter 7 of the thesis we performed a large-scale neural network architecture search for regular and transprecision models. We conclude that transprecision concepts are scalable, fast and easy to obtain, such that they outperform all regular choices.

9.1 Summary of main results

The abstraction of transprecision computing.

We defined the concept of transprecision computing in Chapter 4 as a system that is based on configurable precisions that produces results with a measurable performance at a measurable quality. The quality characterization $Q(\theta)$ and the performance characterization $P(\theta)$ asses precision related system settings. Understanding the user's requirements in connection with a full system characterization allows the delivery of an optimized solution. The more relaxed quality requirements are, the larger the optimization potential. The stricter the quality requirements, the smaller are performance gains. In the worst case, one is forced to conclude that the original baseline application was already the optimal solution. However, we successfully demonstrated that transprecision computing achieves performance gains without quality degenerations at all. For example, PageRank iterations iteratively correct quantization errors performed in early iterations, see Section 5.2.1. Deep learning models classify all samples as in the baseline, even though much smaller data types were used for all the stored weight parameters, see Section 6.2.2. Even in scientific computations, such as in GLQ, knowing that the core routine operates

with limited dynamic range allows reduction of the exponent encoding with no effect on the final quality, see Section 5.2.3.

Approximate computing techniques are case-specific.

We summarized established approximate computing techniques, including loop perforation, task skipping, memoization, and stochastic computing in Chapter 3. We concluded that stochastic computing relies on many key building blocks that are manually designed and mapped to low-level hardware implementations. The fundamental requirement of substantial fractions of customized hardware hinders stochastic computing to become a major general-purpose technique. Additionally, the stochastic nature turns debugging, implementing, and integration of the technique in existing algorithms or frameworks difficult. Additionally, emulations run slowly since they are forced to collect a large number of statistics to provide reliable conclusions. The stated reasons explain the success of the technique on specific problem instances but also highlight the challenges that one needs to overcome to make stochastic computing a serious candidate for general-purpose comping.

We observed similar limiting factors that turn related work interesting but limited to specifically considered cases. Some techniques, such as using multiple inexact rely on the availability of alternative implementations for modular building blocks. They do not contribute to implementing elementary kernels.

Evaluations of approximate computing techniques are often not directly comparable since they act on different input data, or results are obtained at different operation points, for example with quality degeneration of less than 10%-points. Those settings limit the insights on how methods would perform on new data occurring in novel algorithms or applications.

Numerical emulation is the key to assess quality.

To follow a modular and general approach, transprecision requires the evaluation of the final quality depending on transprecision settings. Using emulation allows the addressing of numerical results before developing full hardware systems. To allow for high flexibility, we

defined floatx, a minimally intrusive C++ header-only library that emulates reduced precision as explained in detail in Section 5.1. The high-performance implementation allows the scaling of numerical experiments. We applied the library directly to BLSTM in Section 5.2.2 where the reference is provided as plain C model. Section 6 integrates floatx into PyTorch [62] to provide modular support of numerical evaluations inside a well-established deep learning framework. The integration of floatx into PyTorch powered the subsequent studies. Additionally, the modular approach allows reusing the numerical libraries on new use-cases involving different data and different models.

Deep learning models are error-resilient.

In our thesis, we reproduced the evidence that deep learning models are inherently error-resilient. Prior work demonstrates the error-resilience of deep learning in various aspects. For example, Hill et al. [184] performed a design study that compares reduced fixed-point representations with reduced floating-point representations. They conclude that the extended dynamic range of reduced floating-point representations outperforms fixed-point representations of the same width. FPGA designs, such as the original implementation of BLSTM [182], include in-depth numerical results and operate with small fixed-point implementations causing only minor quality degenerations. Various numerical libraries have been proposed and applied to deep learning instances [166]. Other prior work, such as Xnor-net [204] follows a drastic approach of collapsing multiplications to binary operations. The corner-case of binary number representations indicates the error-resilience of deep neural networks.

We contribute to state-of-the-art in three ways. First, related work is limited in the setting used to evaluate reduced precision. To that end, we evaluated 30 well-established networks in Section 6.2.2, to provide general insights that are consistent among models. Second, our numerical study includes full-grid elaborations that provide insights into numerical behavior. Numerical studies at that detail-level are easy to interpret and reveal the true break-down behavior of models, as presented in Figure 5.3 for BLSTM. That information allows judging guard-bands for specific operation points that provide insights into model robustness. Third, we provide comparisons with

opportunity costs. To that end, we answer the question of what happens if we would differently construct smaller, and henceforth less accurate models. Section 8.3 presents that for a given quality constraint transprecision models outperform models that are alternatively obtained by performing a NAS for the same constraints running with regular 32-bit floating-point formats.

Transprecision computing for algorithms.

We introduced PageRank, BLSTM, and GLQ as representative applications of Big-Data, deep learning, and scientific computing domain in Section 2.1. We used floatx to provided extensive numerical results in Section 5.2.1. We observed that in all three cases, the external setting matters for evaluations. All three algorithms are data-dependent and the specific problem instances affect metrics. Additionally, hyperparameters of control loop provide excellent tuning opportunities as explained in Section 3.3.

Allowing for minor quality degenerations allows for higher performance improvements. For example, allowing PageRank to stop at moderate residual error reduces the number of iterations until convergence, see Section 5.2.1. Additionally, it increases the fraction of the number of reduced precision iterations of the remaining iterations. This positively affects overall performance. Similarly, allowing for minor degenerations allows operating BLSTM and GLQ computations with further reductions in number representations.

Most state-of-art of approximate computing evaluations rely on minor quality degenerations to provide performance improvements. In contrast, we demonstrated, that transprecision computing is capable of delivering zero-quality loss results at conservative operation points of applications. First, the iterative nature of PageRank allows recovering quantization errors obtained at previous iterations. Second, BLSTM profits from the traditional error-resilience of deep learning, see Section 5.2.2. Third, GLQ uses a limited dynamic range in its computations which allows computing with a reduced exponent field to deliver the same quality, see Section 5.2.3.

The practical impact of transprecision computing.

Three practical aspects lead to the success of transprecision computing in general-purpose computing. First, it must be simple enough to generalize to relevant use-cases. Second, it must be effective to remove computational bottlenecks. Third, the process of integration, evaluation, and optimization must be fast enough.

We have demonstrated the concept on three specific applications in Section 5.2.1. For the domain of deep learning, we have extended results to 30 established reference models in Section 6.2.2. Additionally, in Section 8.3, we applied transprecision to over 3,000 models. All cases deliver consistent numerical results. We conclude that reduced precision is general enough to exploit the error-resilience of various deep learning models.

Transprecision results are effective. For example, in deep learning, even in the strictest case of achieving a zero-loss solution, the number representations can be compressed by 2.6× on average as demonstrated in Section 6.2.2.

Transprecision results are obtained by testing various configurations and finding the optimal settings. Since configurations spaces are large, the optimization of the configuration becomes a time-critical task. We contribute on four fronts in accelerating the configuration search. First, the developed floatx library is performance-critical for emulations. The floatx library is lightweight, follows a slim memory footprint, and is performant as explained in Section 5.1.8. Second, floatx compiles on GPUs enabling a simple and performant integration in PyTorch [62] to evaluate models. Third, on the algorithmic-level, we explained why it is beneficial to use the extrinsic over the intrinsic emulation approach. We compared worst-case error bounds and provide empirical results that support our findings as presented in Section 6.1.3. Fourth, we demonstrated in Chapter 7 that heuristics replace full searches and still provide good results. Our contributions enable to perform a single configuration search for one deep learning model within seconds.

9.2 Outlook and future work

We introduced the concept of transprecision computing and we see multiple extensions for future work.

Numerical libraries

In this work, we presented the common number representations. We implemented floatx, a reduced precision library for IEEE 754 [116] conform representations. We followed a modular approach to use floatx as component-wise integration in PyTorch [62] and various applications. However, we see the potential of alternative representations that should be explored similarly in future work. For example, the LNS as introduced in Section 4.1.3 provides different representations that can cover similar dynamic ranges and precisions as traditional floating-point realizations. Hardware units that implement 32-bit float equivalent LNS implementations exist [71]. Critical components consist of look-up tables and interpolation approaches. Hardware units of those performance-critical parts excellently scale with smaller representations due to the exponential dependence of LUTs on the input address space. Additionally, having multiple numerical libraries that emulate different representations allows finding commonalities among them. We expect that the numerical behavior of LNS follows similar error patterns as measured with floatx since reduced representation obeys similar dynamic ranges and precision levels. Exploiting correlations among results obtained with different number systems allows reusing numerical behavior observed with floatx.

Extending the concept to new domains

We posed the question of how future general-purpose computing systems could be improved. We delivered empirical evidence that transprecision computing brings the key aspect to succeed in that task. We demonstrated the concept by studying three applications stemming from three domains. We extended our insights and generalized and scaled statements for deep learning. Joining multiple aspects, such as the integration of floatx in PyTorch [62], scaling numerical studies in

the number of models, and the number of datasets is domain-specific efforts. Additionally, the development of a constraint NAS is an autoML-specific contribution of our work. Nevertheless, understanding research advances in a domain are important to understand alternatives. They act as an opportunity-cost measure to better judge the success of transprecision computing. We think to raise transprecision computing to the next level, the most promising strategy is to apply it to closed domains. For example, we think that weather simulations fit that case due to the following. First, numerical weather simulations based on consortium for small-scale modelling (COSMO) [205, 206] follow iterative patterns where the numeric behavior can be exploited. Second, variations in measured data and the awareness of stochastic behavior in the domain allow interpreting numerical behavior caused by transprecision computing. Due to the noise awareness, the quality of the global behavior of the model can be judged without strict requirements on the arithmetic of partial results. We expect that this design freedom reveals error-resilience at many stages. Third, weather simulations are relevant for many applications such as pollen forecast [207], local wind gust prediction [208], or assessing wind power predictions [209], among many more.

Software tools

In this work, we have demonstrated all the fundamental steps required to build and profit from an effective transprecision computing system. We covered numerical emulation, at low-level, at mid-level, and at the application-level to assess the quality of configurations. We see many opportunities for building software tools around that use-case that simplifies the overall workflow. To demonstrate the practical value, the integration of transprecision computing in existing applications must be easy and quick. We envision an integrated development environment (IDE) with a graphical user interface (GUI) to provide flexibility. The main feature should include loading reference code, annotating variables and routines and automatically characterizing numerical behavior. Due to the dependency on input data, managing data such as loading, modifying, merging, and computing overall statistics becomes a central part of simplifying numerical experiments.

The basics in the form of the logic of the main functionality are already fully covered in this thesis.

Appendix A

Use case: Efficient video classification

Video classification solves the problem of mapping a video clip represented as a sequence of frame-based pixel data to a single label. Video classification poses a challenging task due to the following two reasons: first, the amount of data involved is very large, and second, the classification system has to learn and deal with temporal and spatial aspects of the underlying data. The total workload caused by a classifier varies due to different design choices of composing methods to a full classification system. We evaluate three state-of-the-art neural-network-based approaches for large-scale video classification, where the computational efficiency of the inference step is of particular importance due to the ever-increasing amount of data throughput for video streams. Our evaluation focuses on finding good efficiency vs. accuracy tradeoffs by evaluating different network configurations and parametrizations. In particular, we investigate the use of different temporal subsampling strategies and show that they can be used to effectively trade computational workload against classification accuracy. Using a subset of the YouTube-8M dataset, we demonstrate that workload reductions in the order of 10×, 50× and 100× can be achieved with accuracy reductions of only 1.3%, 6.2% and 10.8%, respectively. Our results show that temporal subsampling is a simple

and generic approach that behaves consistently over the considered classification pipelines and which does not require retraining of the underlying networks.

A.1 Related work

Traditional video classification approaches [210–212] based on global video descriptors have demonstrated success on a variety of datasets. First, interest points are localized and visual features are locally extracted, then the information is compressed to a constant length global descriptor. Second, a standard classifier (e.g., SVM) predicts the final class. However, such approaches often involve the tedious design of hand-crafted features.

Recent methods improve by adopting a data-driven approach where also the features are learned. Naturally, video classification builds the 3D extension of image classification and henceforth, 3D extensions of traditional convolutional NNs (convolutional neural networks (CNNs)) are explored [213, 214]. However, this approach does not scale well to long videos and has thus only been applied to short video clips with a length in the order of seconds. Ng et al. [21] show that different feature pooling architectures employing LSTM [215] are better suited to video classification. However, end-to-end training approaches become infeasible for very large datasets as the YouTube-8M. It contains 50 years of video footage, and even trivial sequential processing of the individual frames becomes a demanding task. Finding good NN configurations is non-trivial due to manual tuning and many learning and validation cycles, which may take weeks or even months. E.g., assuming that a single GPU allows processing at a rate of $\sim$4.3 fps[1], one pass through the 5.8 M training videos of YouTube-8M with an average of 230 frames/video requires around 9.8 years. Renting a cloud infrastructure with multiple GPUs allows to cut the total amount of time but the required costs are in the order of 75 000 $[2]. The estimated cost of complete end-to-end training with the full dataset and a typical amount of 100-1000

[1] Based on the time required for one forward/backward pass for GoogLeNet using Nervana Systems' neon library on an Nvidia GeForce Titan X (GPU) (GTX) Titan X (Maxwell) GPU `https://github.com/soumith/convnet-benchmarks`.

learning epochs is therefore around 7.5 M $-75 M $, which is infeasible for most institutions. To this end, the baseline methods published together with the YouTube-8M dataset split the classification problem into two steps. First, frames are mapped to feature vectors with a lower dimension using a state-of-the-art image classification network (Inception-v3). Second, the classification is performed on the feature vectors. This approach has the advantage that pure data-driven learning is still possible and computational costs remain tractable.

Fast inference methods, such as Low-Rank Expansions [216] reach speedups $< 5\times$ at accuracy drops $< 1\%$. XNOR-Nets [204] gain $58\times$ in speed at 12.5% accuracy reduction on ImageNet. Even though we solve the more complex problem of video classification, our approach is on par with Low-Rank Expansions and outperforms the XNOR-Nets trade-off.

A.2 Practical video classification systems

Our approach targets a large-scale machine learning system based on the two-step approach introduced together with the YouTube-8M dataset. For practical scalability reasons, we avoid end-to-end learning approaches since the raw videos of that dataset amount to a data volume of $\sim$ 1 Petabyte. We evaluate two-step approaches where frame feature vectors are obtained by first extracting features using a CNN trained on an image classification task (without the final classification layer), before applying principle component analysis (PCA) to reduce the dimensionality to 1024 per frame and feature vector. These feature vectors are then used for video classification in a second step. Such a decomposition of the classification problem allows a practical intermediate representation of the video and the handling of the training of two separate subproblems. The vectors have been precomputed using the Inception-v3 CNN and are available as part of the YouTube dataset, which is convenient from a practical viewpoint as it enables us to perform evaluations without having to evaluate the complete CNN.

In our evaluation, we consider three classification pipelines:

[2]Based on a 650 $/month rent for a high-end GPU `https://aws.amazon.com/ec2/instance-types/p2/`.

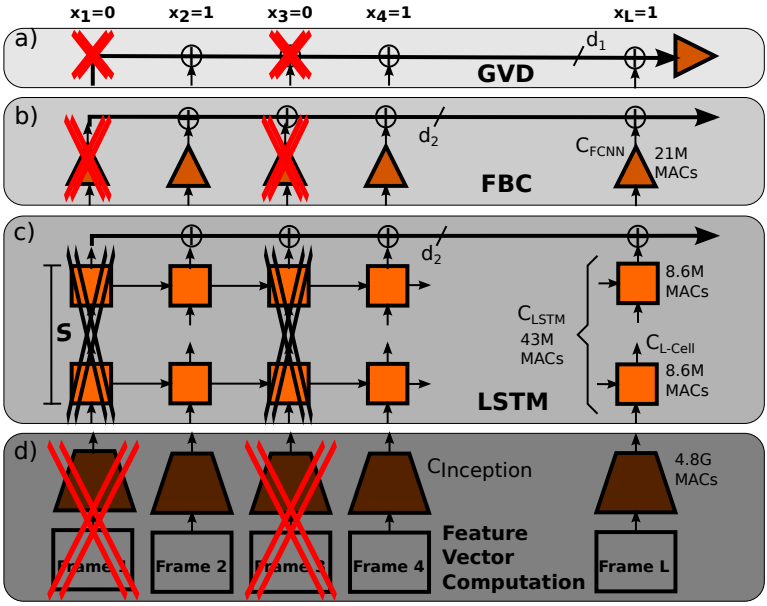


Figure A.1: Overview of the three video classification approaches. First, an inference with the Inception-v3 network (d) produces frame level feature vectors. Second, GVD (a), FBC (b) and LSTM (c) are applied to classify the video. Frames i are skipped if the corresponding decision vector entries $\mathbf{x}_i$ are 0. This is illustrated with red crosses for a regular temporal subsampling of $2\times$.

a. Feature vector aggregation [20], that produces a *global video descriptor* (**global video descriptor (GVD)**) which is then classified using a fully-connected neural network (FCNN).

b. *Frame-based classification* (**frame based classification (FBC)**) [20] with an FCNN, followed by an aggregation of the classification results.

c. A *long short-term memory* (**LSTM**) architecture [21] processes the complete vector sequence of the video.

The three classification system variants are illustrated in Fig. A.1, and only differ in the way the 1024 dimensional feature vectors are processed, as explained in the following.

A.2.1 Global video descriptor (GVD)

As noted by the authors of YouTube-8M [20], GVD exhibits three advantages: fixed-length vectors allow to apply standard learning, the amount of data involved in learning is reduced by the average length of the video (230×), and the approach is generic. As a baseline, we compute the video average feature vector μ as mean over the feature sequence. Since the provided feature vectors are already normalized, no further normalization is required and the resulting GVD μ is fed directly into an FCNN for final classification. The chosen network configurations for this evaluation are listed in Table A.1. The FCNN has 1024 input dimensions, followed by two dense layers that end in H_1 and H_2 neurons, each using ReLU activations which are defined element-wise as $x \mapsto \max(0, x)$. The third, final layer outputs a softmax-normalized prediction for each class obtained by $\mathbf{x} \mapsto e^{\mathbf{x}} / \|e^{\mathbf{x}}\|_1$.

A.2.2 Frame based classification (FBC)

This approach maps each per-frame feature vector to a video class prediction, and all predictions of a video sequence are then aggregated to form the final classification. Learning of the FCNN is achieved by using the global label of a particular video sequence as the local ground-truth label for all frames in that video. The final classification is obtained either by average or max-voting aggregation of the per-frame results. We observed that averaging consistently outperforms a max-voting aggregation and thus use the averaging scheme henceforth.

A.2.3 Long short-term memory (LSTM)

LSTM cells [215] are used to implement a recurrent neural network (RNN) [217], that enables learning of high-level concepts from the temporal information of the input sequence. Long-update chains in

Table A.1: Hyper-parameter configurations and caused workloads in terms of million MAC operations per building block. FCNN configurations are used in both, the GVD and FBC pipelines.

FCNN	k=0	1	2	3	4	5	6	7
H_1	4096	4096	1024	512	256	128	64	32
H_2	1024	4096	1024	512	256	128	64	32
C_{FCNN}	8.4	21	2.1	0.8	0.33	0.15	0.07	0.03
LSTM	k=0	1	2	3	4	5	6	7
H	32	64	128	512	1024	32	64	128
S	1	1	1	1	1	2	2	2
C_{LSTM}	0.02	0.05	0.16	2.21	8.62	0.03	0.09	0.32
LSTM	k=8	9	10	11	12	13	14	
H	512	1024	32	64	128	512	1024	
S	2	2	5	5	5	5	5	
C_{LSTM}	4.42	17.2	0.08	0.23	0.79	11.0	43.0	

RNN architectures may cause vanishing or exploding gradient problems [218]. LSTMs can mitigate that problem by having an internal state storing long temporal information, whereas the rest of the structure (*input*, *output* and *forget* gates) is modeled to capture local (short-term) effects. Table A.1 lists the hyper-parameter choice used in our evaluations. H denotes the dimensionality of the involved hidden state and S is the amount of stacked LSTM-cell chains over the full sequence. The LSTM configuration $k = 13$ corresponds to the proposed configuration from [21] and $k = 9$ refers to the configuration used in [20].

A.3 Computational complexity

The number of MAC operations reliably estimates the inference time for a wide class of CNNs [219]. Hence, we state the workload for one video inference as:

$$\begin{aligned} C_{\mathrm{GVD}} &= L/r \cdot (C_{\mathrm{Inception}} + d_1) + C_{\mathrm{FCNN}_k}, \\ C_{\mathrm{FBC}} &= L/r \cdot (C_{\mathrm{Inception}} + C_{\mathrm{FCNN}_k} + d_2), \\ C_{\mathrm{LSTM}} &= L/r \cdot (C_{\mathrm{Inception}} + C_{\mathrm{LSTM}_k} + d_2), \end{aligned} \tag{A.1}$$

where L refers to the length of the video sequence, $d_1 = 1024$ is the dimension of the used feature vectors, $d_2 = 20$ the number of classes, $C_{\mathrm{Inception}} = 4.8 \cdot 10^9$ the workload to compute the feature vectors, C_{FCNN_k} and C_{LSTM_k} the FCNN configurations according Table A.1, S the number of stacked layers and r the temporal subsampling factor that will be introduced in more detail in the next section. Due to the involved orders of magnitude, we have that $C_{\mathrm{Inception}} \gg C_{\mathrm{FCNN}_k}$, $C_{\mathrm{Inception}} \gg C_{\mathrm{LSTM}_k}$ and $C_{\mathrm{Inception}} \gg d_{1,2}$. Therefore, we observe that $C_{\mathrm{GVD}} \approx C_{\mathrm{FBC}} \approx C_{\mathrm{LSTM}} \approx L/r \cdot C_{\mathrm{Inception}}$.

A.4 Temporal subsampling

More than 99% of the inference workload is caused by the first frame-level feature extraction step, irrespective of the actual classifier chosen. It is, therefore, crucial to introduce optimizations leading to fewer computations in the first step.

There are two common approaches to achieve this. First, a more efficient network architecture than Inception-v3 could be sought by exploring different network topologies, parametrizations and quantizations such as in [204]. Second, a more efficient CNN evaluation engine could be built by employing specialized hardware [220]. While the first approach requires several long training and validation cycles, the second approach requires custom hardware design which is a time-consuming task.

In this chapter, we explore a third approach, where temporal subsampling is used to significantly reduce the amount of CNN evaluations in the first step. Note that this is orthogonal to the other two approaches and does not involve any expensive training iterations

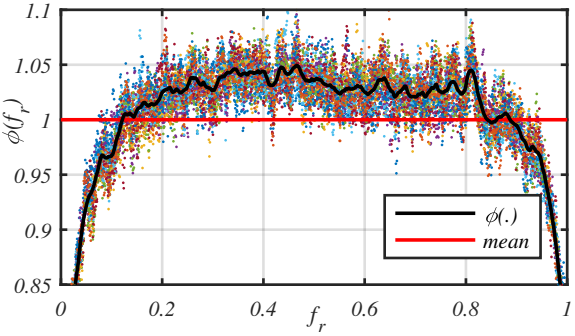


Figure A.2: Estimated probability density $\phi(f_r)$ that a single frame alone is sufficient to correctly classify the complete video sequence. *Prior-based subsampling* improves accuracy by using $\phi(f_r)$ to bias sampling points to frames that are beneficial for classification.

of the large CNN in the first step. In this chapter, we use the full length of the videos to train the three architectures described in Sec. A.2, and introduce temporal subsampling during inference to save computational complexity. Reducing the workload of the inference step is especially critical in large-scale datacenter applications where video footage is being uploaded at an ever-increasing rate[3]. We define a decision function $d(.)$ that outputs a Boolean vector $\mathbf{x} \in \{0,1\}^L$ determining whether a given frame at index i in the video sequence is processed ($x_i = 1$) or skipped ($x_i = 0$). A skipped frame allows us to save a full CNN evaluation in the first step, and the computational workload decreases linearly with the amount of skipped frames.

Subsampling Strategies

We consider two different instances of d termed *regular subsampling* and *prior-based subsampling*, as described below. *Regular subsampling* $d_{\text{reg}}(L, r)$ reduces the total amount of frames L by a fixed factor r in a uniform manner and has the advantage that it can be applied at almost no cost. *Prior-based subsampling* leverages the observation

[3]For example, several hundred video hours are uploaded to YouTube every minute `https://fortunelords.com/youtube-statistics/`.

Table A.2: Dataset used for video classification.

	Original	Training set	Validation set
Number of classes	4 800	20	20
Videos per class	2229†	500	100
Total videos	8 264 650	10 000	2000
Average video length	229.6	227.8	228.2
Number of frames	$1.9 \cdot 10^9$	2 277 717	456 427
Feature data volume	~ 1 PB	~ 9 GB	~ 2 GB

† Average value, videos/entity $\in [120, 539\,926]$ [20].

that the sampling positions of the selected frames have a significant impact on the overall classification outcome.

We consider the conditional probability $p(f \text{ sufficient}|f_r)$ given the relative frame position $f_r \in [0, 1]$ that a single frame f is sufficient to correctly classify the full video sequence. We empirically estimate $\phi(f_r) = \text{Interpolate}(\text{Histogram}(f \text{ is sufficient}))$ as interpolated aggregation of normalized histograms where correct classification results from Section A.2.2 are binned according f_r. Figure A.2 shows that frames taken from the middle of the video sequences are more likely to lead to correct classification outcomes. This is probably due to the fact that most videos contain lead-in and lead-out portions. The prior-based subsampling strategy $d_{\text{prior}}(L, r, \phi)$ introduces this a-priory knowledge by randomly drawing $\lfloor L/r \rfloor$ frame positions from a distribution that is obtained by scaling $\phi(f_r)$ to match the sequence length L.

A.5 Evaluation and results

Section A.5 gives more details on the employed dataset, Section A.5 and Section A.5 explain the training procedure and Section A.6 finally states the comparison of achieved accuracy versus workload tradeoffs.

Subset of YouTube-8M

To shorten the development cycle and make the evaluation practicable, we constructed a subset of the YouTube-8M dataset, a multi-labeled dataset with an average 1.8 of entity labels per video. We

filter out all multi-labeled videos to create a single label problem. Table A.2 states our choice, we assumed as practical resource constraint a data volume limit that fits on a single GPU (Nvidia GeForce GTX Titan X, 12GB random access memory (RAM)). Our subset allows hyper-parameter tuning and multiple repetitions of the full training to state the variance caused due to the random NN initialization in a reasonable time. Note that this runtime reduction approach is suitable in our case since we are not interested in maximizing the absolute classification accuracy, but in the relative accuracy degradation behaviour due to temporal subsampling. The feature vectors provided in the YouTube dataset have been sampled at a rate of 1Hz (one vector per second of video) and the subsampling factor r used in this paper is relative to that rate.

FCNN Training

We used the Adam optimizer [221] to train the FCNNs by minimizing the average of the cross-entropy with a learning rate of 10^{-4}. Note that both the GVD and FBC pipelines employ the same FCNN configurations listed in Table A.1. A batch size of 200 input samples is employed, and the initialization of the weights is achieved using random normal distributed values with a standard deviation of 0.1. The bias values are initialized with a constant offset of 0.1 to break the symmetry. Before the start of each epoch, all input samples are shuffled with a uniform distribution over all possible permutations of the input samples. Training is run for 100 epochs.

LSTM Training

Similar to [20], we unrolled the LSTM for 60 iterations and trained on sequences of 60 consecutive frames at a random offset within the video for 380 epochs. The loss function weighs all individual frame losses with increasing values starting from $1/N$ for the first frame, up to 1 for the last frame in order to enhance learning and classification performance as in [20, 21]. Predictions are computed by considering the full-length video. During training, we used a dropout factor of 0.5.

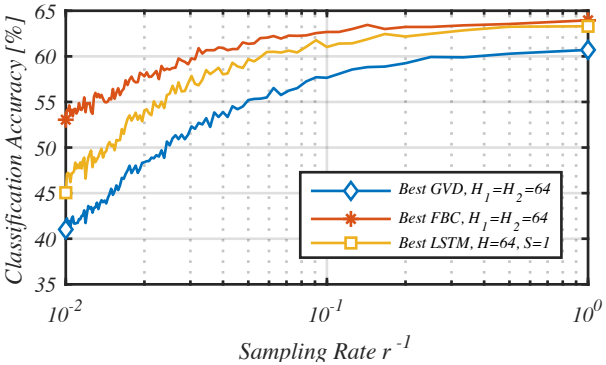


Figure A.3: Comparison of accuracy achieved with best configurations of GVD, FBC and LSTM approach as function of subsampling.

A.6 Results and discussion

In all experiments, the subsampling factor has been swept over the range $1\times$ to $100\times$. The accuracy is measured as Topk rate with $k = 1$, i.e., the number of correct classified videos relative to all classified videos.

Figure A.3 compares the accuracy behavior of the three approaches depending on *regular subsampling*. We explain the stronger decay in the case of the LSTM approach by the fact that LSTM was explicitly trained to learn temporal information which is partially destroyed by subsampling. Still, the best configurations remain ordered as a function of subsampling.

Figure A.4 shows the obtained classification accuracy versus computational workload for the GVD (a), FBC (b) and LSTM (c) approaches using *regular subsampling*. Even though different NNs and different classification systems are considered, the characteristic caused by subsampling is consistent which demonstrates the generality of the approach. In the range where $r \leq 5$ the accuracy degradation is negligible. For larger r factors, subsampling gradually decreases the classification accuracy. Interestingly, FBC seems to be more robust against subsampling than the other two approaches. The best LSTM and GVD configurations lose around 8% of classification

accuracy from $r = 50\times$ to $r = 100\times$, whereas FBC only loses 5%. Figure A.4 d), e) and f) show the additional accuracy improvements (in terms of percentage differences) achieved when employing *prior-based subsampling* instead of *regular subsampling*. Positive values are in favor of *prior-based subsampling*. Noise levels appear more pronounced due to the zoomed-in view. Leveraging prior knowledge in aggressively subsampled regions allows to improve the accuracy of around 2% for the GVD and FBC approaches, and around 4% for the LSTM at negligible extra cost.

A.7 Summary and conclusion

We considered three video classification approaches that employ a two-step classification procedure and presented accuracy and computational complexity results for various NN configurations. Our evaluations show that, while the LSTM-based pipeline outperforms a simple GVD approach, similar classification accuracies as achieved by the LSTM approach can be reached by aggregating FBC results. The FBC method has the advantage that it is considerably easier to train than an LSTM approach.

Further, we note that most of the computational burden stems from the first step involving the evaluation of a large CNN, and investigate temporal subsampling as a simple yet effective way to trade computational complexity against classification accuracy. We introduce two different subsampling strategies and show that significant workload reductions in the order of $10\times$ can be achieved with negligible impact on classification accuracy. Larger workload reductions up to $100\times$ are possible, leading to accuracy degradations in the order of 10% for the best NN configurations.

Temporal subsampling is a simple and generic approach that does not require retraining of large CNNs, and which behaves consistently over the considered classification pipelines. It is straightforward to implement, and therefore well suited for static or dynamic load balancing and cost optimization in large-scale, industrial video classification systems.

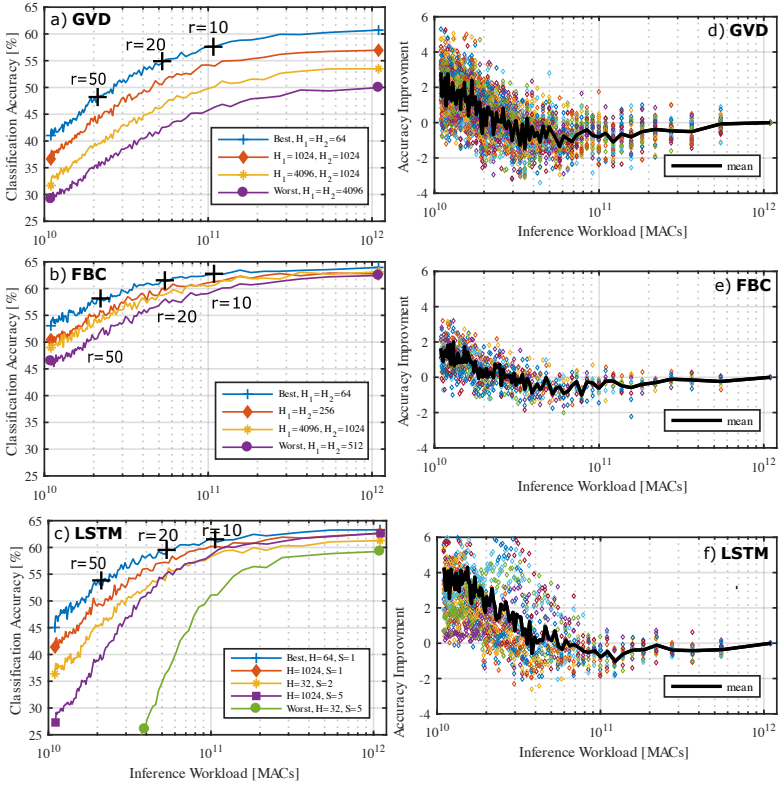


Figure A.4: Comparison of accuracy versus workload tradeoffs achieved with *regular subsampling* for different configurations for the a) GVD b) FBC and c) LSTM approach. For each approach, a selection of representative curves is shown, comprising the best, the worst, and two in-between configurations from Table A.1. Temporal subsampling allows us to effectively trade computational workload against classification accuracy, thereby generating a wide range of Pareto optimal operating points. Improvements due to *prior-based subsamplig* over *regular subsampling* are quantified in d-f) for the three approaches.

Appendix B

Notation and acronyms

Symbols

$\mathbf{A}$	matrix
$\mathbf{a}$	vector
a, A	scalars
$\mathbb{N}$	set of natural numbers
$\mathbb{Z}$	set of integer numbers
$\mathbb{R}$	set of real numbers
$\mathbb{R}^n$	set of real n vectors
$\mathbb{R}^{n \times m}$	set of real $n \times m$ matrices
$\mathcal{T}$	machine learning task
$\mathcal{D}$	machine learning dataset
$\mathcal{D}_S$	source dataset
$\mathcal{D}_T$	target dataset
$\mathcal{A}$	machine learning algorithm
$\mathcal{M}$	machine learning model
$Q_{n,f}$	fixed-point representation of n-bits with f fractional bits
$Q_{n,f}^{unsigned}$	unsigned fixed-point representation

$Q_{n,f}^{signed}$	signed fixed-point representation
$T_{w,t}$	IEEE 754 floating-point representation of w exponent bits and t significand bits
$L_{n,f}$	LNS floating-point representation of n-bits with f fractional bits
q	quality (represented as scalar)
p	performance (represented as scalar)
Θ	configuration space
θ	a specific configuration $\theta \in \Theta$
θ_{opt}	the optimal configuration
θ^*	a good configuration
$(p, q)_\theta$	performance p and quality q for configuration θ

Operators

a_i	i-th entry of vector $\mathbf{a}$
$\mathbf{a}_i$	i-th column of matrix $\mathbf{A}$
$\mathbf{A}_i$	i-th row of matrix $\mathbf{A}$
$\mathbf{A}_{i,j}$	entry of the i-th row and j-th column of matrix $\mathbf{A}$
$(\cdot)^T$	matrix transposition
$\lvert \cdot \rvert$	absolute value
$\lceil \cdot \rceil$	ceil: smallest integer value equal to or larger as argument
$\lfloor \cdot \rfloor$	floor: largest integer value equal to or smaller as argument
$\lVert \cdot \rVert_1$	ℓ^1-norm, i.e., $\sum_i^n \lvert x_i \rvert$ for $\mathbf{x} \in \mathbb{R}^n$
$\lVert \cdot \rVert$	ℓ^2-norm or Euclidean norm, i.e., $\sqrt{\sum_i^n \lvert x_i \rvert^2}$ for $\mathbf{x} \in \mathbb{R}^n$
$\lVert \cdot \rVert_\infty$	ℓ^∞-norm, i.e., $\sum_i^n \max\{\lvert x_i \rvert\}$ for $\mathbf{x} \in \mathbb{R}^n$
$<$	less than
$\leq$	less or equal than
$\ll$	much less than
$>$	greater than
$\geq$	greater or equal than
$\gg$	much greater than
$\succ$	Pareto dominant over (i.e., better in all aspects)
$f(\cdot)$	scalar function

$f'(\cdot)$	first derivative of f
$f''(\cdot)$	second derivative of f
$\exp(\cdot)$	natural exponential function
$\log_2$	base-2 logarithm
$\log_{10}$	base-10 logarithm
$\sum$	summation
$\prod$	product sequence
$\int$	integration
$O(\cdot)$	big-O, asymptotic complexity bound
$P(\cdot)$	Characterization of performance
$Q(\cdot)$	Characterization of quality
$R_{P,Q}(\cdot)$	Characterization of trade-off between performance versus quality
$i{+}{+}$	single step increment, $i \leftarrow i + 1$
$i{+}{=}s$	increment and assign, $i \leftarrow i + s$

Acronyms

AI	artificial intelligence
ALU	arithmetic logic unit
ASIC	application-specific integrated circuit
BLAS	basic linear algebra subprograms
BLSTM	bidirectional LSTM
BN	batch normalization
BW	backward
CNN	convolutional neural network
COSMO	consortium for small-scale modelling
CPU	central processing unit
CSR	compressed sparse row
CTC	connectionist temporal classification
CUDA	compute unified device architecture

DAG	directed acyclic graph
DL	deep learning
DNAS	differentiable neural architecture search
DNN	deep neural network
DOT	dot product
DPN	dual-path networks
DRAM	dynamic random access memory
DSP	digital signal processor
DT	data types
FBC	frame based classification
FCNN	fully-connected neural network
FEM	finite element method
FGPA	field-programmable gate array
floatx	reduced precision floating-point library
FLOP	floating-point operations
FMA	fused-multiply-add
FPGA	field programmable gate array
FPU	floating-point unit
FSM	finite state machine
FW	forward
GAN	generative adversarial network
GEMM	general matrix matrix multiplication
GLQ	Gauss-Legendre quadrature
GNU	GNU's not unix!; free software
GPU	graphical processing unit
GTX	GeForce Titan X (GPU)
GUI	graphical user interface
GVD	global video descriptor
HDL	hardware description language
HDMI	high definition multimedia interface
HPC	high performance computing

HPCG	high performance conjugate gradient
HPO	hyperparameter optimization
HW	hardware
ICDM	international conference on data minin
IDE	integrated development environment
ILSVRC	imagenet - large scale visual recognition challenge
INTLAB	INTerval LABoratory
IoT	internet of things
IPV	inexact program version
KNN	K-nearest neighbors
LAN	local area network
LDPC	low density parity check
LFSR	linear feedback shift register
LINPACK	linear algebra package
LLVM	low level virtual machine
LNS	logarithmic number system
LP	loop perforation
LSTM	long-short-term memory
LUT	look-up table
MAC	multiply-accumulate
ML	machine learning
MNIST	modified national institute of standards and technology
MPFR	multiple precision floating-point reliably
NaN	not a number
NAS	neural architecture search
NN	neural network
OCR	optical character recognition
ONNX	open neural network exchange

PCA	principle component analysis
PULP	parallel ultra-low-power
RAM	random access memory
RNN	recurrent neural network
RTL	register-transfer level
SC	stochastic computing
SFINAE	substitution failure is not an error
SGD	stochastic gradient descent
SIMD	single instruction multiple data
SIMT	single instruction multiple threads
SNG	stochastic number generator
SPEC	standard performance evaluation corporation
SVM	support vector machines
SW	software
TAPAS	train-less accuracy predictor for architecture search
TP	transprecision
TSM	task skipping and memoization
UNUM	universal number
USB	universal serial bus
VGG	visual geometry group

Bibliography

[1] M. M. Waldrop, "More than moore," *Nature*, vol. 530, no. 7589, pp. 144–148, 2016.

[2] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer networks*, vol. 56, no. 18, pp. 3825–3833, 2012.

[3] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1390–1395.

[4] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1105–1110.

[5] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 62, 2016.

[6] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*. IEEE, 2013, pp. 1–6.

[7] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[8] D. Mladenić, J. Brank, M. Grobelnik, and N. Milic-Frayling, "Feature selection using linear classifier weights: interaction with classification models," in *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2004, pp. 234–241.

[9] Z. Deng, X. Zhu, D. Cheng, M. Zong, and S. Zhang, "Efficient knn classification algorithm for big data," *Neurocomputing*, vol. 195, pp. 143–148, 2016.

[10] A. Pradhan, "Support vector machine-a survey," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 8, pp. 82–85, 2012.

[11] V. Y. Kulkarni and P. K. Sinha, "Pruning of random forest classifiers: A survey and future directions," in *2012 International Conference on Data Science & Engineering (ICDSE)*. IEEE, 2012, pp. 64–68.

[12] L. Wolf, T. Hassner, and I. Maoz, "Face Recognition in Unconstrained Videos with Matched Background Similarity," in *IEEE CVPR*, 2011.

[13] G. Zhao, X. Huang, M. Taini, S. Z. Li, and M. Pietikäinen, "Facial Expression Recognition From Near-Infrared Videos," *Image and Vision Computing*, vol. 29, no. 9, 2011.

[14] L. Xia, C.-C. Chen, and J. Aggarwal, "View Invariant Human Action Recognition using Histograms of 3D Joints," in *IEEE CVPRW*, 2012.

[15] F. Caba Heilbron, V. Escorcia, B. Ghanem, and J. Carlos Niebles, "ActivityNet: A Large-Scale Video Benchmark for Human Activity Understanding," in *IEEE CVPR*, June 2015.

[16] K. G. Derpanis, M. Lecce, K. Daniilidis, and R. P. Wildes, "Dynamic Scene Understanding: The Role of Orientation Features in Space and Time in Scene Classification," in *IEEE CVPR*, 2012, pp. 1306–1313.

[17] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild," *CoRR*, vol. abs/1212.0402, 2012.

[18] O. Kliper-Gross, T. Hassner, and L. Wolf, "The Action Similarity Labeling Challenge," *IEEE TPAMI*, vol. 34, no. 3, pp. 615–621, 2012.

[19] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale Video Classification with Convolutional Neural Networks," in *IEEE CVPR*, 2014.

[20] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8M: A Large-Scale Video Classification Benchmark," *arXiv:1609.08675*, 2016.

[21] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond Short Snippets: Deep Networks for Video Classification," in *IEEE VCPR*, June 2015.

[22] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[23] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[25] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[26] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng, "Dual path networks," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach,

R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4467–4475. [Online]. Available: http://papers.nips.cc/paper/7033-dual-path-networks.pdf

[27] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[28] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[29] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy *et al.*, "Chapter 15 - evolving deep neural networks," in *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, R. Kozma, C. Alippi, Y. Choe, and F. C. Morabito, Eds. Academic Press, 2019, pp. 293 – 312. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128154809000153

[30] L. Xie and A. Yuille, "Genetic cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1379–1388.

[31] Z. Zhong, J. Yan, and C. Liu, "Practical network blocks design with q-learning," *CoRR*, vol. abs/1708.05552, 2017. [Online]. Available: http://arxiv.org/abs/1708.05552

[32] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *CoRR*, vol. abs/1611.01578, 2016. [Online]. Available: http://arxiv.org/abs/1611.01578

[33] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[34] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[35] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *CoRR*, vol. abs/1611.02167, 2016. [Online]. Available: http://arxiv.org/abs/1611.02167

[36] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 2902–2911.

[37] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[38] M. Wistuba, "Deep learning architecture search by neuro-cell-based evolution with function-preserving mutations," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 243–258.

[39] R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, and A. C. I. Malossi, "Tapas: Train-less accuracy predictor for architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 3927–3934.

[40] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *arXiv preprint arXiv:1905.01392*, 2019.

[41] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," *CoRR*, vol. abs/1807.11626, 2018. [Online]. Available: http://arxiv.org/abs/1807.11626

[42] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware

efficient convnet design via differentiable neural architecture search," *CoRR*, vol. abs/1812.03443, 2018. [Online]. Available: http://arxiv.org/abs/1812.03443

[43] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[44] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[45] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[47] N. L. Roux, P. antoine Manzagol, and Y. Bengio, "Topmoumoute online natural gradient algorithm," in *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, Eds. Curran Associates, Inc., 2008, pp. 849–856. [Online]. Available: http://papers.nips.cc/paper/3234-topmoumoute-online-natural-gradient-algorithm.pdf

[48] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[49] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[50] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[51] P. Singh, "Bayesian optimization for machine learning," Tech. rep., San Diego State University, Tech. Rep., 2018.

[52] G. Malkomes and R. Garnett, "Automating bayesian optimization with bayesian optimization," in *Advances in Neural Information Processing Systems*, 2018, pp. 5984–5994.

[53] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *arXiv preprint arXiv:1603.06560*, 2016.

[54] G. Mariani, F. Scheidegger, R. Istrate, C. Bekas, and C. Malossi, "Bagan: Data augmentation with balancing gan," *arXiv preprint arXiv:1803.09655*, 2018.

[55] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, Oct 2010.

[56] W.-Y. Chen, Y.-C. Liu, Z. Kira, Y.-C. F. Wang, and J.-B. Huang, "A closer look at few-shot classification," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=HkxLXnAcFQ

[57] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *CoRR*, vol. abs/1512.00567, 2015.

[58] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.

[59] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE CVPR*, 2009, pp. 248–255.

[60] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner, "United states data center energy usage report," 06/2016 2016.

[61] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[62] "Pytorch," https://pytorch.org/, accessed: 2019-05-22.

[63] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, 2015, pp. 1–6.

[64] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[65] F. Scheidegger, L. Benini, C. Bekas, and C. Malossi, "Constrained deep neural network architecture search for iot devices accounting for hardware calibration," in *Advances in Neural Information Processing Systems*, 2019, to appear.

[66] G. Flegar, F. Scheidegger, V. Novakovic, G. Mariani, A. Tomas, C. Malossi, and E. Quintana-Ortí, "Float x: A c++library for customized floating-point arithmetic," *ACM Trans. Math. Softw.*, 2019, to appear.

[67] F. Scheidegger, L. Cavigelli, M. Schaffner, A. Malossi, C. Bekas, and L. Benini, "Impact of temporal subsampling on accuracy and performance in practical video classification," in *Signal Processing Conference (EUSIPCO), 2017 25th European.* IEEE, 2017, pp. 996–1000.

[68] F. Scheidegger, R. Istrate, G. Mariani, L. Benini, C. Bekas, and C. Malossi, "Efficient image dataset classification difficulty estimation for predicting deep-learning accuracy," *arXiv preprint arXiv:1803.09588*, 2018.

[69] R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, and A. C. I. Malossi, "Tapas: Train-less accuracy

predictor for architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 3927–3934.

[70] A. Sood, B. Elder, B. Herta, C. Xue, C. Bekas, A. C. I. Malossi, D. Saha, F. Scheidegger, G. Venkataraman, G. Thomas *et al.*, "Neunets: An automated synthesis engine for neural network design," *arXiv preprint arXiv:1901.06261*, 2019.

[71] Y. Popoff, F. Scheidegger, M. Schaffner, M. Gautschi, F. K. Gürkaynak, and L. Benini, "High-efficiency logarithmic number unit design based on an improved cotransformation scheme," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1387–1392.

[72] M. Schaffner, F. Scheidegger, L. Cavigelli, H. Kaeslin, L. Benini, and A. Smolic, "Towards edge-aware spatio-temporal filtering in real-time," *IEEE Transactions on Image Processing*, vol. 27, no. 1, pp. 265–280, 2018.

[73] M. Eggimann, C. Gloor, F. Scheidegger, L. Cavigelli, M. Schaffner, A. Smolic, and L. Benini, "Hydra: An accelerator for real-time edge-aware permeability filtering in 65nm cmos," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–5.

[74] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Journal*, vol. 19, no. 1, pp. 43–49, 1976.

[75] E. H. Sibley, "Dhrystone: A synthetic systems," *Communications of the ACM*, vol. 27, no. 10, 1984.

[76] K. M. Dixit, "The spec benchmarks," *Parallel computing*, vol. 17, no. 10-11, pp. 1195–1209, 1991.

[77] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[78] B. Subramaniam, W. Saunders, T. Scogland, and W.-c. Feng, "Trends in energy-efficient computing: A perspective from the green500," in *2013 International Green Computing Conference Proceedings*. IEEE, 2013, pp. 1–8.

[79] J. Dongarra, M. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems. university of tennessee," *Electrical Engineering and Computer Sciente Department, Technical Report UT-EECS-15-736*, 2015.

[80] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[81] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip *et al.*, "Top 10 algorithms in data mining," *Knowledge and information systems*, vol. 14, no. 1, pp. 1–37, 2008.

[82] A. Y. Ng, A. X. Zheng, and M. I. Jordan, "Stable algorithms for link analysis," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2001, pp. 258–266.

[83] T. Haveliwala, "Efficient computation of pagerank," Stanford, Tech. Rep., 1999.

[84] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen, "Efficient pagerank approximation via graph aggregation," *Information Retrieval*, vol. 9, no. 2, pp. 123–138, 2006.

[85] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub, "Extrapolation methods for accelerating pagerank computations," in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 261–270.

[86] M. R. Yousefi, M. R. Soheili, T. M. Breuel, E. Kabir, and D. Stricker, "Binarization-free ocr for historical documents using lstm networks," in *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 2015, pp. 1121–1125.

[87] C. Patel, D. Shah, and A. Patel, "Automatic number plate recognition system (anpr): A survey," *International Journal of Computer Applications*, vol. 69, no. 9, 2013.

[88] I. Z. Yalniz and R. Manmatha, "A fast alignment scheme for automatic ocr evaluation of books," in *2011 International Conference on Document Analysis and Recognition*, Sep. 2011, pp. 754–758.

[89] M. H. Dave and M. Patel, "A survey on handwritten character recognition using multilayer perceptron neural networks," 2016.

[90] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[91] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm networks," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 4. IEEE, 2005, pp. 2047–2052.

[92] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 369–376.

[93] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[94] M. G. Larson and F. Bengzon, "The finite element method: theory, implementation, and practice," *Texts in Computational Science and Engineering*, vol. 10, 2010.

[95] K. Yamazaki and A. Abe, "Loss analysis of interior permanent magnet motors considering carrier harmonics and magnet eddy currents using 3-d fem," in *2007 IEEE International Electric Machines & Drives Conference*, vol. 2. IEEE, 2007, pp. 904–909.

[96] C. Joulin, J. Xiang, J.-P. Latham, and C. Pain, "A new finite discrete element approach for heat transfer in complex shaped multi bodied contact problems," in *International Conference on Discrete Element Methods*. Springer, 2016, pp. 311–327.

[97] M. Vahab and N. Khalili, "Numerical investigation of the flow regimes through hydraulic fractures using the x-fem technique," *Engineering Fracture Mechanics*, vol. 169, pp. 146–162, 2017.

[98] G. Castellazzi, A. M. D'Altri, S. de Miranda, and F. Ubertini, "An innovative numerical modeling strategy for the structural analysis of historical monumental buildings," *Engineering Structures*, vol. 132, pp. 229–248, 2017.

[99] F. G. Lether, "On the construction of gauss-legendre quadrature rules," *Journal of Computational and Applied Mathematics*, vol. 4, no. 1, pp. 47–52, 1978.

[100] V. I. Krylov and A. H. Stroud, *Approximate calculation of integrals*. Courier Corporation, 2006.

[101] R. W. Freund and R. W. Hoppe, *Stoer/Bulirsch: Numerische Mathematik 1*. Springer-Verlag, 2007.

[102] K. Shivaram, "Generalised gaussian quadrature over a triangle," *American Journal of Engineering Research*, vol. 2, no. 09, pp. 290–293, 2013.

[103] A. Genz, "A package for testing multiple integration subroutines," in *Numerical Integration*. Springer, 1987, pp. 337–340.

[104] A. Rahimi, L. Benini, and R. K. Gupta, "Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, 2016.

[105] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67–73, 2004.

[106] K. Palem and A. Lingamneni, "Ten years of building broken chips: The physics and engineering of inexact

computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 87:1–87:23, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465789

[107] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *Proceedings of the 9th Conference on Computing Frontiers*, ser. CF '12. New York, NY, USA: ACM, 2012, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/2212908.2212912

[108] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 3, p. 9, 2014.

[109] A. Teman, G. Karakonstantis, R. Giterman, P. Meinerzhagen, and A. Burg, "Energy versus data integrity trade-offs in embedded high-density logic compatible dynamic memories," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 489–494. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2755864

[110] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 124–134.

[111] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, July 2005.

[112] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 13–24.

[113] A. Raha and V. Raghunathan, "Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart

camera system," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[114] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 394–406.

[115] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, "Approxma: Approximate memory access for dynamic precision scaling," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 337–342.

[116] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[117] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *International Static Analysis Symposium*. Springer, 2011, pp. 316–333.

[118] S. Misailovic, D. M. Roy, and M. Rinard, "Probabilistic and statistical analysis of perforated patterns," 2011.

[119] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 35–50.

[120] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 198–209.

[121] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 92:1–92:19, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465794

[122] A. Klein, "Linear feedback shift registers," in *Stream Ciphers*. Springer, 2013, pp. 17–58.

[123] H. Krawczyk, "Lfsr-based hashing and authentication," in *Advances in Cryptology — CRYPTO '94*, Y. G. Desmedt, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 129–139.

[124] T. E. Tkacik, "A hardware random number generator," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 450–453.

[125] G. G. Lorentz, *Bernstein polynomials*. American Mathematical Soc., 2013.

[126] D. Zhang and H. Li, "A stochastic-based fpga controller for an induction motor drive with integrated neural network algorithms," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 2, pp. 551–561, 2008.

[127] W. J. Gross, V. C. Gaudet, and A. Milner, "Stochastic implementation of ldpc decoders," in *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.* IEEE, 2005, pp. 713–717.

[128] Z. Zhang, V. Anantharam, M. J. Wainwright, and B. Nikolic, "An efficient 10gbase-t ethernet ldpc decoder design with low error floors," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 4, pp. 843–855, 2010.

[129] J. A. Dickson, R. D. McLeod, and H. Card, "Stochastic arithmetic implementations of neural networks with in situ learning," in *IEEE International Conference on Neural Networks*. IEEE, 1993, pp. 711–716.

[130] B. D. Brown and H. C. Card, "Stochastic neural computation. i. computational elements," *IEEE Transactions on computers*, vol. 50, no. 9, pp. 891–905, 2001.

[131] B. D. Brown and H. C. Card, "Stochastic neural computation. ii. soft competitive learning," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, 2001.

[132] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[133] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[134] M. L. Overton, *Numerical computing with IEEE floating point arithmetic*. Siam, 2001, vol. 76.

[135] J. Detrey and F. De Dinechin, "A tool for unbiased comparison between logarithmic and floating-point arithmetic," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 49, no. 1, pp. 161–175, 2007.

[136] M. Schaffner, M. Gautschi, F. K. Gürkaynak, and L. Benini, "Accuracy and performance trade-offs of logarithmic number units in multi-core clusters," in *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. IEEE, 2016, pp. 95–103.

[137] M. Gautschi, M. Schaffner, F. K. Gürkaynak, and L. Benini, "An extended shared logarithmic unit for nonlinear function kernel acceleration in a 65-nm cmos multicore cluster," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 98–112, 2016.

[138] T.-J. Kwon, J. Sondeen, and J. Draper, "Design trade-offs in floating-point unit implementation for embedded and processing-in-memory systems," in *2005 IEEE International Symposium on Circuits and Systems*. IEEE, 2005, pp. 3331–3334.

[139] K. Karuri, R. Leupers, G. Ascheid, H. Meyr, and M. Kedia, "Design and implementation of a modular and portable ieee 754 compliant floating-point unit," in *Proceedings of the Design*

Automation & Test in Europe Conference, vol. 2. IEEE, 2006, pp. 1–6.

[140] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on computers*, vol. 60, no. 7, pp. 913–922, 2010.

[141] N. G. Kingsbury and P. J. Rayner, "Digital filtering using logarithmic arithmetic," *Electronics Letters*, vol. 7, no. 2, pp. 56–58, 1971.

[142] E. E. Swartzlander and A. G. Alexopoulos, "The sign/logarithm number system," *IEEE Transactions on Computers*, vol. 100, no. 12, pp. 1238–1242, 1975.

[143] J. N. Coleman, C. I. Softley, J. Kadlec, R. Matousek, M. Tichy, Z. Pohl, A. Hermanek, and N. F. Benschop, "The european logarithmic microprocesor," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 532–546, 2008.

[144] R. C. Ismail and J. Coleman, "Rom-less lns," in *2011 IEEE 20th Symposium on Computer Arithmetic*. IEEE, 2011, pp. 43–51.

[145] J. Coleman and R. C. Ismail, "Lns with co-transformation competes with floating-point," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 136–146, 2015.

[146] M. Gautschi, M. Schaffner, F. K. Gürkaynak, and L. Benini, "4.6 a 65nm cmos 6.4-to-29.2 pj/flop@ 0.8 v shared logarithmic floating point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2016, pp. 82–83.

[147] K. Deb and H. Gupta, "Searching for robust pareto-optimal solutions in multi-objective optimization," in *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 2005, pp. 150–164.

[148] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library

with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 13, 2007.

[149] S. M. Rump, "Intlab-interval laboratory, developments in reliable computing," in *the Proc. SCAN-98 conference, Kuluwer Academic Publishers, the Netherland*, 1999, pp. 77–104.

[150] V. Lefèvre, "Sipe: Small integer plus exponent," in *2013 IEEE 21st Symposium on Computer Arithmetic*. IEEE, 2013, pp. 99–106.

[151] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[152] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[153] J. Hauser, "Berkeley softfloat project home page," http://www.jhauser.us/arithmetic/SoftFloat.html, accessed: September 2019.

[154] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1051–1056.

[155] T. L. Veldhuizen, "C++ templates are turing complete," *Available at citeseer. ist. psu. edu/581150. html*, 2003.

[156] S. A. Figueroa, "When is double rounding innocuous?" *ACM SIGNUM Newsletter*, vol. 30, no. 3, pp. 21–26, 1995.

[157] É. Martin-Dorel, G. Melquiond, and J.-M. Muller, "Some issues related to double rounding," *BIT Numerical Mathematics*, vol. 53, no. 4, pp. 897–924, Dec 2013. [Online]. Available: https://doi.org/10.1007/s10543-013-0436-2

[158] S. M. Rump, "Ieee754 precision-k base-β arithmetic inherited by precision-m base-β arithmetic for k< m," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 3, p. 20, 2017.

[159] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Transactions on Mathematical Software (TOMS)*, vol. 17, no. 3, pp. 410–423, 1991.

[160] S. Boldo and G. Melquiond, "Emulation of a fma and correctly rounded sums: Proved algorithms using rounding to odd," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 462–471, 2008.

[161] ISO, "ISO international standard ISO/IEC 14882:2017(E) – Programming Language C++," https://isocpp.org/std/the-standard. Visited June 2018., 2017.

[162] H. Anzt, J. Dongarra, and E. S. Quintana-Ortí, "Adaptive precision solvers for sparse linear systems," in *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*. ACM, 2015, p. 2.

[163] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas, "Finding authorities and hubs from link structures on the world wide web," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 415–429.

[164] P. Hill, B. Zamirai, S. Lu, Y.-W. Chao, M. Laurenzano, M. Samadi, M. Papaefthymiou, S. Mahlke, T. Wenisch, J. Deng *et al.*, "Rethinking numerical representations for deep neural networks," 2016.

[165] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," *arXiv preprint arXiv:1711.10374*, 2017.

[166] D. M. Loroch, F.-J. Pfreundt, N. Wehn, and J. Keuper, "Tensorquant: A simulation toolbox for deep neural network quantization," in *Proceedings of the Machine Learning on HPC Environments*, ser. MLHPC'17. New York, NY, USA: ACM, 2017, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/3146347.3146348

[167] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The german traffic sign recognition benchmark: A multi-class classification competition," in *The 2011 International Joint Conference on Neural Networks*, July 2011, pp. 1453–1460.

[168] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[169] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. DudÃk, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 215–223. [Online]. Available: http://proceedings.mlr.press/v15/coates11a.html

[170] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," in *NIPS workshop on deep learning and unsupervised feature learning*, vol. 2011, no. 2, 2011, p. 5.

[171] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva, and A. Torralba, "Places: A 10 million image database for scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[172] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, and A. Vedaldi, "Describing textures in the wild," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 3606–3613. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2014.461

[173] L. Bossard, M. Guillaumin, and L. Van Gool, "Food-101 – mining discriminative components with random forests," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 446–461.

[174] M. E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *2008 Sixth Indian Conference on Computer Vision, Graphics Image Processing*, Dec 2008, pp. 722–729.

[175] A. Quattoni and A. Torralba, "Recognizing indoor scenes," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, pp. 413–420.

[176] W. Li, T. Logenthiran, V.-T. Phan, and W. L. Woo, "A novel smart energy theft system (sets) for iot based smart home," *IEEE Internet of Things Journal*, 2019.

[177] G. Fenza, M. Gallo, and V. Loia, "Drift-aware methodology for anomaly detection in smart grid," *IEEE Access*, vol. 7, pp. 9645–9657, 2019.

[178] M. M. Gaber, A. Aneiba, S. Basurra, O. Batty, A. M. Elmisery, Y. Kovalchuk, and M. H. U. Rehman, "Internet of things and data mining: From applications to techniques and systems," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 3, p. e1292, 2019.

[179] A. Nordrum, "The internet of fewer things [news]," *IEEE Spectrum*, vol. 53, no. 10, pp. 12–13, October 2016.

[180] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," *CoRR*, vol. abs/1802.03268, 2018. [Online]. Available: http://arxiv.org/abs/1802.03268

[181] Y. Weng, T. Zhou, L. Liu, and C. Xia, "Automatic convolutional neural architecture search for image classification under different scenes," *IEEE Access*, vol. 7, pp. 38 495–38 506, 2019.

[182] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 1394–1399.

[183] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *CoRR*, vol. abs/1405.3866, 2014. [Online]. Available: http://arxiv.org/abs/1405.3866

[184] P. Hill, B. Zamirai, S. Lu, Y. Chao, M. Laurenzano, M. Samadi, M. C. Papaefthymiou, S. A. Mahlke, T. F. Wenisch, J. Deng *et al.*, "Rethinking numerical representations for deep neural networks," *CoRR*, vol. abs/1808.02513, 2018. [Online]. Available: http://arxiv.org/abs/1808.02513

[185] L. Cavigelli and L. Benini, "Extended bit-plane compression for convolutional neural network accelerators," *CoRR*, vol. abs/1810.03979, 2018. [Online]. Available: http://arxiv.org/abs/1810.03979

[186] A. Ashiquzzaman, L. V. Ma, S. Kim, D. Lee, T. Um, and J. Kim, "Compacting deep neural networks for light weight iot scada based applications with node pruning," in *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, Feb 2019, pp. 082–085.

[187] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[188] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[189] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[190] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive

neural architecture search," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[191] S. Xie, R. Girshick, P. Dollar, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[192] "Train cifar10 with pytorch," https://github.com/kuangliu/pytorch-cifar, accessed: 2019-05-22.

[193] G. Flegar, F. Scheidegger, V. Novakovic, G. Mariani, A. Tomas, C. Malossi, and E. Quintana-Ortí, "Float x: A c++library for customized floating-point arithmetic," 2019, submitted.

[194] "Raspberry pi 3 model b+ product description," https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/, accessed: 2019-05-14.

[195] S. Mittal, "A survey on optimized implementation of deep learning models on the nvidia jetson platform," *Journal of Systems Architecture*, 2019.

[196] "Raspberry pi celebrates 25 millionth sale as 7th anniversary arrives," https://www.tomshardware.com/news/raspberry-pi-25-million-sold,38724.html, accessed: 2019-05-22.

[197] "Onnx: Open neural network exchange format," https://onnx.ai/, accessed: 2019-05-22.

[198] "Arm neon," https://www.arm.com/why-arm/technologies/neon, accessed: 2019-05-22.

[199] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *Foundations of genetic algorithms*. Elsevier, 1991, vol. 1, pp. 69–93.

[200] X. Y. Stella and J. Shi, "Multiclass spectral clustering," in *null*. IEEE, 2003, p. 313.

[201] F. Scheidegger, R. Istrate, G. Mariani, L. Benini, C. Bekas, and C. Malossi, "Efficient image dataset classification difficulty estimation for predicting deep-learning accuracy," 2019, submitted.

[202] F. Conti, D. Rossi, A. Pullini, I. Loi, and L. Benini, "Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision," *Journal of Signal Processing Systems*, vol. 84, no. 3, pp. 339–354, Sep 2016. [Online]. Available: https://doi.org/10.1007/s11265-015-1070-9

[203] 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.

[204] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*. Springer, 2016, pp. 525–542.

[205] P. Steiner, V. Stauch, M. Gwerder, D. Gyalistras, B. Lehmann, M. Morari, and F. Schubiger, "Numerical weather prediction at meteoswiss," in *Proceedings of the 30th meeting of the European Working Group on Limited Area Modelling (EWGLAM) and 15th meeting of the Short Range Numerical Weather Prediction network (SRNWP). Madrid, Spain*, 2008.

[206] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, M. Raschendorfer, and T. Reinhardt, "Operational convective-scale numerical weather prediction with the cosmo model: description and sensitivities," *Monthly Weather Review*, vol. 139, no. 12, pp. 3887–3905, 2011.

[207] K. Zink, H. Vogel, B. Vogel, D. Magyar, and C. Kottmeier, "Modeling the dispersion of ambrosia artemisiifolia l. pollen with the model system cosmo-art," *International journal of biometeorology*, vol. 56, no. 4, pp. 669–680, 2012.

[208] N. Helbig, R. Mott, A. van Herwijnen, A. Winstral, and T. Jonas, "Parameterizing surface wind speed over complex topography," *Journal of Geophysical Research: Atmospheres*, vol. 122, no. 2, pp. 651–667, 2017.

[209] G. Giebel, R. Brownsword, G. Kariniotakis, M. Denhard, and C. Draxl, "The state-of-the-art in short-term prediction of wind power. a literature overview," Jan 2011.

[210] J. Liu, J. Luo, and M. Shah, "Recognizing Realistic Actions from Videos "In The Wild"," in *IEEE CVPR*, 2009, pp. 1996–2003.

[211] J. C. Niebles, C.-W. Chen, and L. Fei-Fei, "Modeling temporal structure of decomposable motion segments for activity classification," in *ECCV*. Springer, 2010, pp. 392–405.

[212] H. Wang, M. M. Ullah, A. Klaser, I. Laptev, and C. Schmid, "Evaluation of Local Spatio-Temporal Features for Action Recognition," in *BMVC*. BMVA Press, 2009.

[213] S. Ji, W. Xu, M. Yang, and K. Yu, "3D Convolutional Neural Networks for Human Action Recognition," *IEEE TPAMI*, vol. 35, no. 1, 2013.

[214] M. Baccouche, F. Mamalet, C. Wolf, C. Garcia, and A. Baskurt, "Sequential Deep Learning for Human Action Recognition," in *HBU*. Springer, 2011, pp. 29–39.

[215] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[216] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[217] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[218] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long-Term Dependencies with Gradient Descent is Difficult," *IEEE TNNLS*, 1994.

[219] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[220] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[221] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980*, 2014.

Curriculum Vitae

Florian Scheidegger was born in Bern, Switzerland, in 1990 and grew up in Matten (BE), Switzerland. He received his B.Sc. and M.Sc. degrees in electrical engineering and information technology from ETH Zürich, Switzerland in 2014 and 2017. He worked for five months at the Integrated Systems Laboratory of ETH developing a spatio-temporal video pipeline in the first part, followed by working in the domain of deep learning. In 2017, he joined IBM to pursue a PhD degree under the supervision of Prof. Dr. Luca Benini and Dr. Cristiano Malossi.