ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

DATA SCIENCE AND COMPUTATION

Ciclo 37

Settore Concorsuale: 09/E3 - ELETTRONICA

Settore Scientifico Disciplinare: ING-INF/01 - ELETTRONICA

LEVERAGING TRANSPRECISION COMPUTING TECHNIQUES FOR ULTRA-LOW-POWER IOT DEVICES

Presentata da: Seyed Ahmad Mirsalari

Coordinatore Dottorato

Daniele Bonacorsi

Supervisore

Luca Benini

Co-supervisore

Giuseppe Tagliavini

Esame finale anno 2026

ALMA MATER STUDIORUM–UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF ELECTRICAL, ELECTRONIC AND INFORMATION ENGINEERING

"GUGLIELMO MARCONI"

37th Cycle Degree in Data Science and Computation

DEI

Thesis in

Leveraging Transprecision Computing Techniques for Ultra-Low-Power IoT devices

CANDIDATE:

Seyed Ahmad Mirsalari

Advisor:

Prof. Luca Benini

Co-Advisor:

Prof. Giuseppe Tagliavini

Final Exam year 2026

ABSTRACT

This dissertation investigates how transprecision computing—the use of multiple numeric precisions across the hardware-software stack—enables real-time machine learning (ML) and digital signal processing (DSP) workloads on Internet of Things (IoT) edge devices based on microcontroller units (MCUs). Motivated by strict energy and memory constraints, the work targets floating-point "smallFloats" formats (FP16/BF16/FP8) as a middle ground between fixed-point and full-precision arithmetic, preserving application-level quality while substantially reducing computation and data footprint. The thesis presents four main contributions: (1) the adoption of FP8 beyond deep learning (DL), (2) TransLib, a transprecision kernel library for embedded multi-core platforms, (3) an optimized Self-Organizing Map (SOM) design for on-device bacterial-genome identification, and (4) StreamEase, an automatic streaming transformation for Temporal Convolutional Networks (TCNs). First, the thesis explores FP8 formats outside the DL domain across classic DSP/ML kernels. With proper exponent allocation and scaling strategies, FP8 achieves acceptable numerical error relative to FP32 while delivering substantial speedups of $3.14\times$-$18.81\times$ on 1-8 cores. Second, TransLib provides a workflow for transprecision design on embedded multi-core systems, supporting datatype exploration, fixed- and mixed-precision mappings, memory-footprint analysis, and an end-to-end heart-rate detection pipeline, enabling systematic tuning of accuracy–performance trade-off. Third, a joint algorithm–architecture co-design exploits FP8 arithmetic to enable ultra-low-power SOM-based genomics. This approach achieves accurate DNA-sequence classification with a $2\times$ memory reduction versus 16-bit representations, speedups up to $18.7\times$ over FP32, and a $6.72\times$ energy-efficiency improvement over FP32 (vs. $3.15\times$ for a CGRA baseline). Finally, StreamEase automatically converts non-streaming TCNs into streaming networks, enabling real-time inference with multi-timestep processing while preserving accuracy. On GAP9, a Conv-TasNet speech-enhancement model reaches 2 ms inference latency—33% of a 6.25 ms budget—with $108.9\times$ fewer MACs and $27.7\times$ fewer cycles than the non-streaming baseline. Collectively, this dissertation provides a hardware-validated methodology for deploying transprecision DSP/ML applications on MCU-class edge devices.

ACKNOWLEDGMENTS

I would first like to express my deepest gratitude to my supervisors, Prof. Luca Benini and Prof. Giuseppe Tagliavini, for their invaluable guidance, constant support, and insightful feedback throughout my Ph.D. journey. Working under their supervision has been an inspiring and enriching experience. In particular, I am profoundly thankful to Giuseppe, whose mentorship, patience, and encouragement gradually transformed him from a great mentor into a great friend.

I would also like to acknowledge the APROPOS Project members, especially Prof. Jari Nurmi, Dr. Aleksandr Ometov, and all the Early Stage Researchers (ESRs). The numerous events, training sessions, and workshops we shared together not only enriched my research experience but also created lasting memories and friendships. My sincere thanks go to the GreenWaves Technologies team for hosting me during my secondment, particularly the Machine Learning team—Léo (Ahmad) Bijar, Marco Fariselli, Martin Croome, and Francesco Paci—for their collaboration and openness. I am especially grateful to Farnaz Legrand, one of the kindest people I met there, for her warmth and generosity. I also wish to thank my collaborators at KTH Royal Institute of Technology, Prof. Ahmed Hemani and Saba Yousefzadeh, for the insightful discussions and productive joint projects we had together.

My gratitude also goes to all my colleagues and friends in Bologna and Zürich for creating a pleasant and supportive environment throughout these years. In particular, my time at Neurolab was made more enjoyable by my labmates, whose positive energy and everyday interactions made the Ph.D. journey much lighter. A special mention goes to my friends from my home country—Mohsen Seyedkazemi Ardebili, Amirhossein Moallemi, Fatemeh (Vida) Bozorgi, Amirhossein Kiamarzi, Mahyar PourJabar, Mohammad Hasan (Mojtaba) Ahmadilivani, and Abolfazl Ghasemzadeh—for making this journey brighter and easier through their friendship and support. I am equally thankful to my flatmates at Casa Amendola, who patiently endured the unpredictable schedule and lifestyle of a Ph.D. student and made those years more enjoyable.

Finally, my deepest appreciation goes to my parents, whose unconditional love, sacrifices, and encouragement have been the foundation of all my achievements, and to my brothers, for their constant support and for always believing in me.

CONTENTS

Abstract i

Contents v

List of Tables vi

List of Figures viii

1 Introduction 1
- 1.1 Adopting Float8 Formats outside the Deep-Learning Domain 3
- 1.2 TransLib: A Library to Explore Transprecision Floating-Point Arithmetic on Multi-Core IoT End-Nodes 4
- 1.3 Optimizing Self-Organizing Maps for Bacterial Genome Identification on Parallel Ultra-Low-Power Platforms 6
- 1.4 StreamEase: Enabling Real-Time Inference of Temporal Convolution Networks on Low-Power MCUs with Stream-Oriented Automatic Transformation 6
- 1.5 Thesis Structure 7

2 Related work 8
- 2.1 Approximate and Transprecision Computing on MCUs 8
- 2.2 ISA Extensions for Transprecision 9
- 2.3 From Transprecision on MCUs to Edge AI and Streaming TCNs 10

3 Background 12
- 3.1 PULP 12
- 3.2 GAP9 13
- 3.3 Transprecision 14
- 3.4 Power and Energy Characterization of PULP 14

4 Adopting FP8 Formats Outside the Deep-Learning Domain 16
- 4.1 Methodology 16

4.1.1 High-level simulations . . . 16
4.1.2 Software library for embedded targets . . . 17
4.2 Experimental Results . . . 18
4.2.1 High-level simulations . . . 19
4.2.2 FP8 hardware support . . . 20
4.3 Conclusion . . . 21

5 TransLib: An End-to-End Workflow for Transprecision Computing 22
5.1 TransLib Design . . . 22
5.1.1 TransLib kernels . . . 24
5.1.2 Generators . . . 24
5.1.3 Accuracy metrics . . . 26
5.1.4 Fused Operations for Mixed-Precision Support . . . 26
5.1.5 Optimized C code . . . 27
5.1.6 Debug support . . . 29
5.1.7 Development flow . . . 29
5.1.8 Automated Precision-Aware Exploration . . . 29
5.2 Hardware platforms . . . 30
5.3 Experimental results . . . 31
5.3.1 Data Type Exploration . . . 31
5.3.2 Fixed-precision on GAP9 . . . 31
5.3.3 Mixed-precision on PULP . . . 35
5.3.4 Memory footprint . . . 36
5.3.5 End-to-End Application: Heart Rate Detection . . . 37
5.4 Conclusion and Future Work . . . 39

6 SOM: A Domain-Specific Application for Reduced-Precision Arithmetic 40
6.1 Related Work . . . 41
6.2 Background . . . 41
6.2.1 SOM algorithm . . . 41
6.3 Methodology . . . 42
6.3.1 Computation Mapping . . . 42
6.3.2 FP8 . . . 43
6.3.3 Vectorization . . . 43
6.3.4 Algorithm and Architectural Optimizations . . . 43
6.4 Experimental Results . . . 44
6.4.1 Experimental Setup . . . 44
6.4.2 Accuracy . . . 45
6.4.3 Hardware Results . . . 45
6.5 Conclusion and Future Work . . . 48

7 StreamEase: Stream-Oriented Conversion of Temporal Convolutional Networks for Edge AI 49
7.1 Methodology . . . 50
7.1.1 Streaming . . . 51
7.1.2 Streaming Transformation Flow . . . 53
7.2 Experimental Results . . . 57
7.2.1 Experimental Setup . . . 57
7.2.2 Accuracy Results . . . 58

7.2.3 Hardware Mapping 58
7.2.4 Comparison with Speech Enhancement solutions on MCUs 60
7.3 Network Architecture: Design Choices and Trade-offs 60
7.4 Conclusion 64

8 Conclusion & Future Work **65**

A Supplementary Material: Public Code Repositories and Optimization Overview **67**
A.1 Public Code Repositories 67
A.1.1 TransLib – A Transprecision Kernel Library for IoT End-Nodes . . . 67
A.1.2 StreamEase – Streaming Conversion for TCNs on Low-Power MCUs 69
A.1.3 SOM-on-PULP – Optimizing Self-Organizing Maps for Bacterial Genome Identification 70
A.2 Summary on Optimization Strategies 71

Bibliography **73**

LIST OF TABLES

3.1 Normalized cycles, energy, and average power for FP32, INT32, and vectorized FP16 on the PULP SoC. 15

4.1 Evaluation of error metrics (MSE, RAE, and accuracy) for MATMUL and CONV kernels in FP16 and the best FP8 configurations for normal distributions with defined mean and standard deviation. 18

5.1 Brief description of TransLib kernels. 23
5.2 Flags used in the golden reference model and optimized C implementations. . 25
5.3 fixed and mixed-precision results relative to the FP32 reference. In the mixed-precision variants, the first two entries denote the data type used for the input data, and the last entry denotes the output type. K-Means kernel only has one input in the mixed-precision: The first entry is the input data type, and the third entry is the output data type. 30
5.4 Execution time of fused mixed-precision (-mfaux) and vectorization (normalized to FP32). In the mixed-precision variants, the first two entries specify the input data type, and the last entry specifies the output type. 35
5.5 Percentage of memory saving compared to float32. In the mixed-precision variants, the first two entries denote the data type used for the input data, and the last entry denotes the output type. 36
5.6 Fixed- and mixed-precision configurations for the PT ECG pipeline. Tuples indicate formats for (pre-proc, filtering, QRS, HR). 37

6.1 Memory required for storing the weights of two SOM neurons in bytes and Area in mm^2 46

7.1 PESQ and memory footprint of Tinydenoiser and TCN denoiser after PTQ using FP16, and Mixed INT8-(B)FP16. 58

7.2 Performance of the non-streaming (NS) and streaming models with different timesteps and quantization on GAP9. The table reports MAC operations (millions), cycles (millions), MAC/Cycle ratio, inference time (ms), and active time (%). 59
7.3 Comparison with Speech Enhancement Solutions on MCUs. 59
7.4 Comparison of different configurations based on the Conv-TasNet architecture. R represents the number of repetitions. X denotes the number of convolutional blocks within each repetition. D is the maximum dilation rate. RF indicates the receptive field measured in seconds. Model sizes are expressed in million parameters. 60

A.1 Comparative Overview of Optimization Techniques Across Kernels 72

LIST OF FIGURES

1.1 High-level roadmap: system constraints lead to smallFloat-based transprecision techniques that map onto MCU-class platforms and enable Edge-AI workloads. Our contributions are the four boxes inside the bottom dashed group, labeled (1)–(4). 4
1.2 Chapter structure after the Introduction 7

3.1 GAP9 architecture. RV denotes RISC-V processor cores; FPU denotes shared floating-point units; L1 TCDM and L2 denote on-chip memory levels. 13
3.2 A comparison of computation methodologies 14

4.1 Analysis of a) random data distribution, b) scaled random data distribution for FP8 kernels. 17
4.2 Effect of varying biases in FP8 formats on MSE for the CONV kernel at a standard deviation of 0.5. 17
4.3 (Speedup of parallel (1, 2, 4, 8 cores), vectorized (FP16/bfloat16, FP8) using 1, 2, 4, and 8 cores relative to the FP32 (1 core) baseline. 18
4.4 Energy improvement of parallel (1, 2, 4, 8 cores), vectorized (FP16/bfloat16, FP8) using 1, 2, 4, and 8 cores relative to the FP32 (1 core) baseline. 19

5.1 Workflow of the TransLib development and evaluation framework. 23
5.2 Details of the data and execution layers in the high-level and the C configuration. Hw flag is the hardware mixed promotion flag. 28
5.3 Speedup of parallel (1, 2, 4, and 8 cores) and vectorized (float16/bfloat16, float8) kernel versions w.r.t. the float32 baseline (1 core). 32
5.4 Energy saving of parallel (1, 2, 4, and 8 cores) and vectorized (float16/bfloat16, float8) kernel versions w.r.t. the float32 baseline (1 core). 34
5.5 ECG processing pipeline: signal conditioning (DC cancel & normalization), filtering & feature extraction (LP, HP, derivative, squaring, MWI), and beat detection & rate estimation (QRS detect $\rightarrow$ RR $\rightarrow$ 60/RR(s)). 37

6.1 Mapping patterns 42

6.2 Example of tiling and double buffering 42
6.3 Speedup of parallel (2, 4, 8 cores), vectorized (FP16/bfloat16, FP8), and CGRA fabric (1, 2, 4, and 8 columns) compared to the FP32 baseline. V and H represent vertical and horizontal mapping, respectively. 46
6.4 Energy improvement of parallel (2, 4, 8 cores), vectorized (FP16/bfloat16, FP8), and CGRA fabric (1, 2, 4, and 8 columns) compared to the FP32 baseline. V and H represent vertical and horizontal mapping, respectively. 46

7.1 End-to-end StreamEase pipeline 50
7.2 Real-time execution of a TCN. The dilation rate of each layer is indicated by D. The receptive field size is 9 timesteps. 51
7.3 Streaming TCN. Dashed black lines represent buffers at timestep $t-1$. Updated buffers (black lines) containing the intermediate outputs at $t-1$ are utilized as input at timestep t. 51
7.4 Real-time cLN process. `ReduceSum`, `CumSum`, and `Expand` normalize inputs by incorporating previous feature statistics. 52
7.5 Streaming cLN at timestep $t-n$. A buffer stores the last `CumSum` outputs, another one tracks the number of previous timesteps for `Expand`. 53
7.6 Algorithm to convert a trained TCN into a streaming network. 54
7.7 Buffers are indicated by red rectangles, and $ts = n$ is the new timestep. The green convolution operations are computed for this timestep. 55
7.8 (a) Non-streaming network with receptive field of 28 timesteps. (b) The streaming network with three input timesteps. The data for processing each timestep is indicated using consistent colors. Since the dilation rate is one (two) in the first (second) layer, there are overlaps between the data for the last two (one) timesteps. These overlaps are not color-coded. 56
7.9 Effect of varying network configuration in Conv-TasNet on buffer size for streaming. 61
7.10 Comparison of reduction rates across different numbers of timesteps in streaming mode compared to non-streaming mode. 62
7.11 Comparison of multi-timesteps approach in non-streaming TCN (NS-TCN) and streaming modes TCN (S-TCN). 63
7.12 MAC per cycle for different network configurations. NS and S represent non-streaming and streaming modes, respectively. 63

CHAPTER

1

INTRODUCTION

Energy efficiency is an essential concern in modern computer systems [21], particularly in the embedded domain, which constitutes about 98% of the computing continuum [18,23]. Despite the developments in semiconductor technology and the advancement of energy-efficient design techniques, the total energy consumption of computer systems is still rising at an unprecedented pace to process an ever-increasing amount of information [135]. Improving the energy efficiency of these emerging workloads is critical to keeping up with the growing amount of data that needs to be processed. Fortunately, most applications are tolerant of approximation and inherently resilient to error [135]. Nowadays, many applications —such as image rendering, signal processing, data mining, robotics, and speech recognition—can tolerate inexact operations and imprecise data in large portions of their execution [21]. In other words, these applications do not require utterly accurate computation, and an "acceptable" result may be substituted for precise outputs [135]. At the cost of some quality degradation, Approximate computing (AxC) offers a substantial efficiency gain for applications that can tolerate inexactness in their output [20, 39, 107, 126, 137].

On power-constrained platforms, fixed-point implementations are a common approach to enhance energy efficiency and performance for applications that compute with real numbers [139]. This approach involves manually adjusting the range and precision of operands based on the processing chain's requirements. However, implementing fixed-point optimizations is complex and demanding, as it requires a comprehensive understanding of the target algorithms [69, 101, 139]. Moreover, applications with strict accuracy and dynamic range requirements pose further challenges when using fixed-point representations. Effective management of normalization and saturation becomes crucial in such cases, as they can increase application development time and introduce overhead due to the need for explicit adaptation of numerical ranges [69].

In scenarios where computational accuracy and a wide dynamic range are critical requirements, adopting floating-point (FP) arithmetic is highly beneficial [101]. FP operations support a broader range of values and deliver significantly higher precision, making them essential for such applications. In high-performance computing, dense linear-algebra kernels (e.g.,

DGEMM in BLAS/LAPACK) and the LINPACK/HPL benchmark that ranks TOP500 supercomputers are defined in double precision [2, 4, 93], and large astrophysical simulations (e.g., GADGET-4) rely heavily on FP64 for stability and dynamic range [2, 4, 93, 131]. Operational weather and climate models (ECMWF's IFS) and CFD solvers such as OpenFOAM use FP extensively and study single- vs double-precision trade-offs for performance vs forecast/solver accuracy [44, 115].

Floating-point formats smaller than 32 bits (smallFloats) reduce memory and bandwidth while keeping floating-point semantics. The most common are IEEE binary16 (FP16), bfloat16 (BF16), and emerging float8 (FP8) encodings. Adopting smallFloats on Internet of Things (IoT) end nodes offers two key benefits for reducing execution time and energy consumption: (i) the reduction of the memory footprint for streaming data and model parameters, and (ii) shorter execution time via exploiting Single Instruction Multiple Data (SIMD) vectorization. Operating simultaneously on multiple sub-word elements in a 32-bit architecture offers a theoretical speedup of up to 2× and 4× for 16-bit and 8-bit data, respectively. This approach reduces the memory footprint by an equivalent amount. Finally, the simultaneous transfer of adjacent data elements makes vectorization more efficient for data movement.

Consistent with these observations, Tagliavini et al. [120] conducted a study showing that utilizing smallFloats can result in considerable energy savings. This outcome is due to the simplification of arithmetic circuitry and the reduction in memory bandwidth requirements for data transfer between memory and registers enabled by vectorization techniques. Motivated by the same push toward reduced precision, recent work has focused on FP8 for both inference and training [86, 92]. These works explored alternative FP8 formats with different trade-offs for exponent and mantissa widths, evaluating their effect under different data and weight distributions [53][1]. A key finding is that FP8 can surpass INT8 in model accuracy with an appropriate allocation of exponent and mantissa bits [53]. However, these studies primarily report accuracy and typically omit a joint evaluation of hardware-centric metrics such as power and latency; likewise, area/power comparisons for FP vs. integer arithmetic are often based on simplified accumulator models (e.g., Kulisch vs. FP accumulators) and thus provide only coarse approximations [130].

In recent years, transprecision computing has matured from a conceptual evolution of approximate computing into a practical methodology that orchestrates precision across the hardware–software stack—at design time and at run time—to reduce energy while preserving application-level quality [80]. On the architecture side, low-precision multiply with higher-precision accumulation is now mainstream (e.g., NVIDIA Hopper FP8 Tensor Cores with FP16/FP32 accumulators) [3], and embedded instruction set architectures (ISAs) expose low-precision FP/SIMD for Embedded microcontroller units (MCUs) (Arm Helium; RISC-V smallFloat SIMD), minimizing conversion overheads while preserving accumulator precision [112, 122]. On the software side, prototypical research toolchains also support mixed-precision tuning and dynamic precision control, highlighting the need for co-design between software analyses and hardware features [17, 26].

Thanks to their flexibility, low power consumption, and low cost, embedded MCUs remain the most common computing platforms for heavily power-constrained, battery-powered IoT end-nodes. In recent years, multi-core platforms have emerged for near-sensor workloads. The open-source PULP (Parallel Ultra-Low Power) project defines an architectural template comprising a host processor and an accelerator composed of general-purpose RISC-V cores that support transprecision techniques [121].

The same pressures that make smallFloats attractive on MCU-class devices—shrinking

[1] https://github.com/Qualcomm-AI-research/FP8-quantization

memory traffic, fitting models into tight on-chip SRAM, and exploiting sub-word SIMD—also make it practical to run Artificial Intelligence (AI) next to the sensor. Edge AI is an innovative computing paradigm that aims to execute Machine Learning (ML) inference (and lightweight adaptation) on the device rather than in a remote data center [74, 118]. Doing so cuts end-to-end latency, avoids fragile network round-trips, and keeps raw data local for privacy and cost reasons [72, 111]. Crucially, many embedded pipelines still benefit from floating-point semantics (e.g., normalization, signal conditioning, on-device learning), so smallFloats (FP16/BF16/FP8) offer a practical middle ground: they retain dynamic range while delivering the energy and bandwidth savings your previous section motivates [53, 76, 122].

Three developments make this shift feasible today. (i) Models and toolchains for tiny platforms (TinyML, MLPerf-Tiny) provide reference tasks and baselines for real-time embedded workloads [11, 134]. (ii) ISA and microarchitecture support brings reduced-precision FP/SIMD to MCUs (Arm Helium/MVE; open RISC-V FP units like FPnew), lowering conversion overheads while preserving higher-precision accumulation where needed [112, 122]. (iii) Mixed-precision practice—compute in smallFloat, accumulate in higher precision—lets designers meet tight energy and memory budgets without abandoning FP behavior [76].

These advances motivate an end-to-end assessment on embedded platforms, where designers must jointly consider accuracy, execution time, and memory footprint constraints. In this context, a transprecision library targeting modern platforms for IoT computing must provide (i) an accuracy model spanning multiple FP types, (ii) SIMD vectorization, and (iii) multi-core parallelism.

As depicted in Fig. 1.1, in this study, the following key aspects are examined:

- Adopting Float8 Formats outside the Deep-Learning Domain
- TransLib: A Library to Explore Transprecision Floating-Point Arithmetic on Multi-Core IoT End-Nodes
- Investigating smallFloats for Bacterial Genome Identification on Parallel Ultra-Low-Power Platforms
- StreamEase: Enabling Real-Time Inference of Temporal Convolution Networks on Low-Power MCUs with Stream-Oriented Automatic Transformation

1.1 Adopting Float8 Formats outside the Deep-Learning Domain

First, the benefits of utilizing the FP8 format beyond deep learning applications are explored. An optimized implementation for the PULP platform based on the RISC-V ISA is presented. The primary objective is to assess how adopting FP8 affects critical metrics, including power consumption, execution time, and memory footprint. The main contributions of this work are the following:

- Investigating the impact of different numbers of bits on the error metrics in different kernels such as FIR, FFT, DWT, Matmul, CONV, SVM, and K-means.
- Introducing algorithmic variants to use smallFloat data types on a parallel computing cluster.
- Enabling vectorization techniques to improve the performance of smallFloat data types

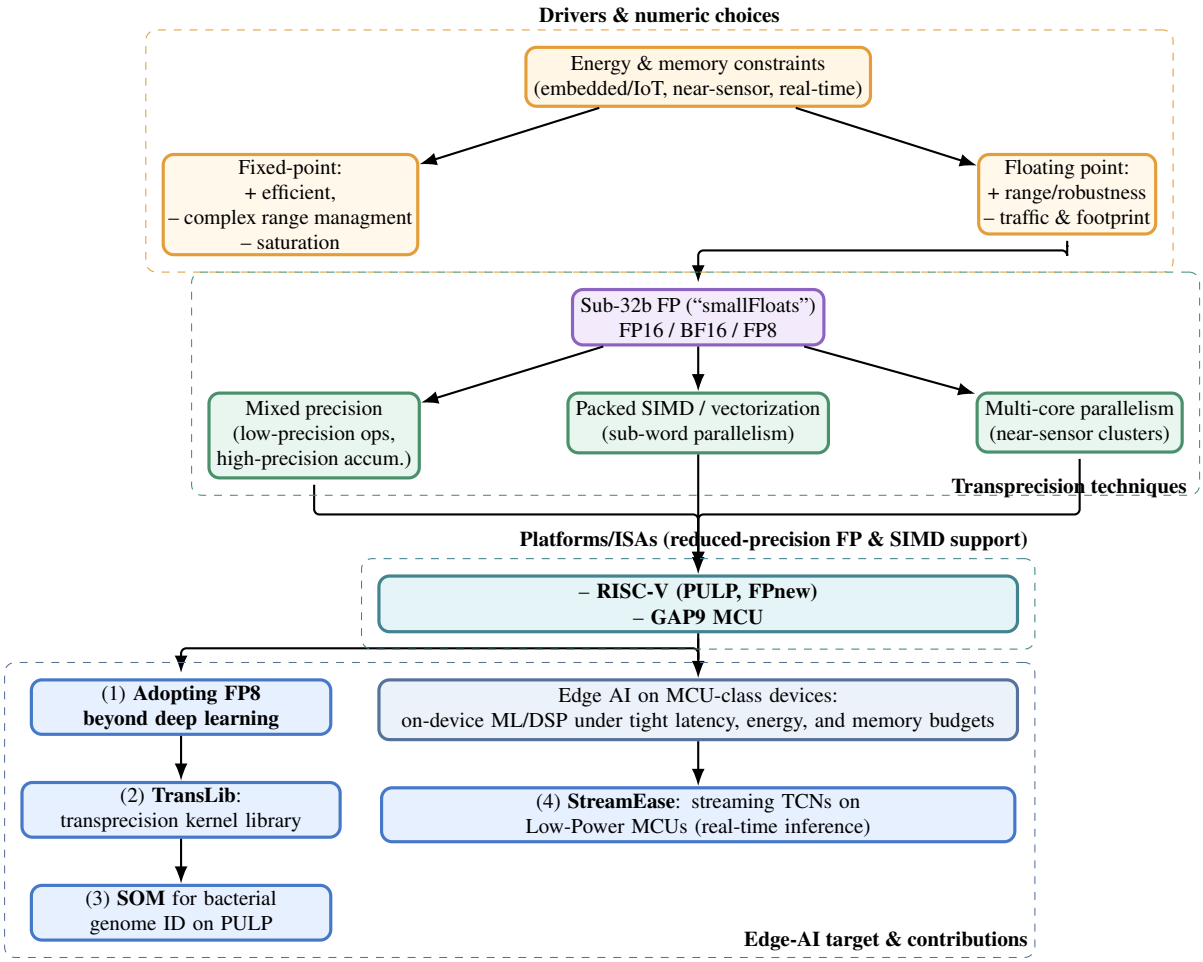


Figure 1.1: High-level roadmap: system constraints lead to smallFloat-based transprecision techniques that map onto MCU-class platforms and enable Edge-AI workloads. Our contributions are the four boxes inside the bottom dashed group, labeled (1)–(4).

- Assessing the impact of smallFloat data types, specifically the FP8 data type, on the accuracy, memory footprint, and performance metrics

The results show that appropriate exponent selection and scaling methods yield acceptable errors compared to FP32. This study paves the way for adopting FP8 formats in non-deep learning domains to achieve consistent energy efficiency and speedup improvements without compromising accuracy. On average, speedup improvements of $3.14\times$, $6.19\times$, $11.11\times$, and $18.81\times$ are observed on 1, 2, 4, and 8 cores, respectively. Furthermore, the vectorized FP8 implementation in the same setup delivers remarkable energy savings, with rates of $2.97\times$, $5.07\times$, $7.37\times$, and $15.05\times$.

1.2 TransLib: A Library to Explore Transprecision Floating-Point Arithmetic on Multi-Core IoT End-Nodes

Second, TransLib[2] is introduced as an open-source kernel library based on transprecision computing principles. TransLib provides knobs to exploit different FP data types (i.e., FP32, FP16,

[2] https://github.com/ahmad-mirsalari/TransLib

bfloat16, FP8), also considering the trade-off between fixed-precision and mixed-precision solutions. The included kernels target two application domains widely represented on IoT end-node devices: signal processing and machine learning. These algorithms are commonly used across different application fields, such as healthcare [40], smart cities [138], computer vision, and image processing [119].

The kernels included in TransLib can be employed in different ways. For hardware designers, TransLib acts as a benchmark suite for functional validation and performance estimation. For application designers, it provides validated building blocks for assessing the adoption of smallFloats in real applications, starting from well-characterized basic kernels.

The main contributions of TransLib are:

- Kernel library and golden models. An open-source library with Python golden models and optimized C implementations targeting IoT end-nodes. Golden models generate datasets/outputs, support fixed and mixed precision (e.g., FP32, FP16, bfloat16, FP8 variants), and include simple hooks to add or swap basic error metrics as needed.
- Precision-aware exploration framework. A systematic framework to evaluate fixed-precision and mixed-precision settings (including FP8 variants), using error metrics such as MAE/MSE/RMSE/RAE.
- Parallel and vector variants. Portable kernels in sequential form, their multi-core versions (2/4/8 cores), and packed-SIMD implementations for smallFloats.
- Mixed-precision scalar variants. Kernel variants adopting fused multiply-and-accumulate (MAC) instructions that multiply low-precision operands with high-precision accumulation (e.g., `FP8×FP8→FP32`, `FP16×FP16→FP32`), reducing conversion overhead and memory traffic.
- Mixed-precision SIMD variants. Kernel variants that leverage fused dot-product-and-sum instructions that compute the dot product between two low-precision vector operands and add the result to a high-precision scalar accumulator, reducing conversion and extraction overheads.
- Evaluation on PULP-Open and GAP9 parallel platforms. Build flows for the PULP-Open and GAP9, including parameterized kernel configurations and scripts to regenerate datasets, metrics, and figures for artifact evaluation. Experimental results quantify speedup, energy consumption, memory footprint, and accuracy, and analyze the impact of vector width, core count, and data type.
- Evaluation in a full application context. A streaming pipeline for heart rate analysis based on the Pan–Tompkins algorithm demonstrates how to evaluate end-to-end trade-offs using the TranLib kernels.

On a single core, enabling packed-SIMD delivers average speedups of about $1.8\times$ with 16-bit data types, and $2.95\times$ with FP8, while reducing the memory footprint by 14–70% vs. FP32, depending on kernel, data type, and problem size. Software-emulated mixed-precision slows down to 0.71–$0.89\times$ compared to fixed-precision, but fused MAC instructions restore this value to $\approx 1.00\times$. Moreover, the fused dot-product-and-sum SIMD path reaches $\approx 2.6\times$ speedup, while numerical error metrics remains low (`FP16` typically MSE $\sim 10^{-6}$–10^{-5}; FP8 improved via `FP8×FP8→FP16` to $\lesssim 7 \times 10^{-3}$).

1.3 Optimizing Self-Organizing Maps for Bacterial Genome Identification on Parallel Ultra-Low-Power Platforms

Third, a case study addressing bacterial genome identification is presented. Given the significant threat posed by pathogenic bacteria to human health, precise and efficient methods for rapid species identification are essential. This work addresses the challenges of executing genomics workloads on embedded platforms with limited computational resources by presenting an optimized implementation of Self-Organizing Maps (SOMs) targeting a parallel ultra-low-power platform. The main contributions of this case study include:

- Proposing two mapping methods for implementing the SOM algorithm on a parallel computing cluster.
- Designing an algorithm to support various network sizes by employing tiling and double buffering to transfer data between different levels of the memory hierarchy.
- Introducing algorithmic variants to use smallFloat data types for the SOM algorithm.
- Enabling vectorization techniques to improve the performance of smallFloat data types.
- Assessing the impact of smallFloat data types, specifically the FP8 data type, on the accuracy, memory footprint, and performance metrics.
- Comparing the results with a coarse-grain reconfigurable architecture (CGRA) fabric that uses a 16-bit fixed-point format.
- Providing an open-source repository[3] that includes the implementation of the SOM algorithm and the integration of the FP8 data type in PyTorch.

Experimental results demonstrate that the FP8-based approach enables accurate DNA sequence classification while reducing memory footprint by a factor of two compared to 16-bit representations. Furthermore, the FP8 implementation demonstrates an impressive speedup over the FP32 implementation, up to $18.7\times$ in large SOM networks. In contrast, the CGRA-based implementation achieves a speedup of $11.05\times$.

Additionally, the FP8 implementation exhibits lower energy consumption than the 16-bit CGRA implementation [136], due to dedicated algorithmic optimizations that achieve a $6.72\times$ improvement in energy efficiency compared to FP32, while the CGRA achieves a $3.15\times$ improvement in large SOM networks.

1.4 StreamEase: Enabling Real-Time Inference of Temporal Convolution Networks on Low-Power MCUs with Stream-Oriented Automatic Transformation

Temporal Convolutional Networks (TCNs) are an effective tool for time-series analysis but face challenges in real-time deployment on resource-constrained devices. Finally, StreamEase[4] is introduced as a Python library that automates the conversion of TCN models to a streaming

[3] https://github.com/ahmad-mirsalari/SOM-on-PULP
[4] https://github.com/ahmad-mirsalari/StreamEase

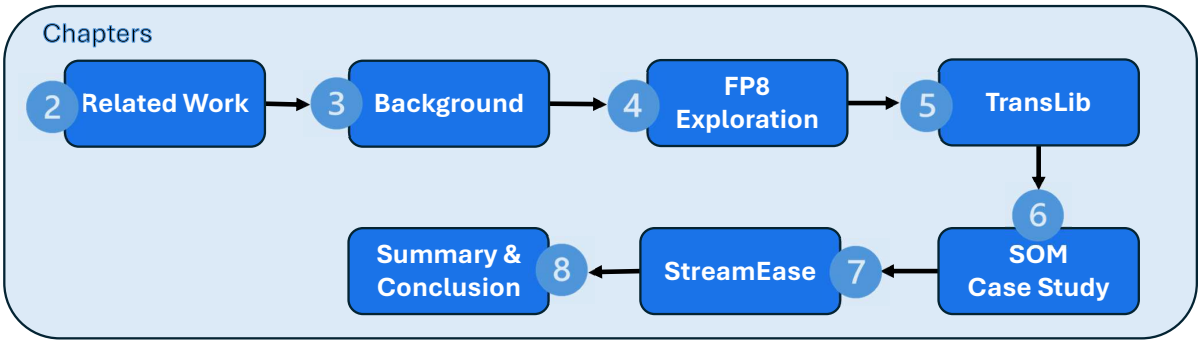


Figure 1.2: Chapter structure after the Introduction

format without affecting model accuracy. This approach significantly reduces deployment time and computational resources. StreamEase enables unified multi-timestep streaming, balancing network versatility and computational demands.

Deploying a speech enhancement model (Conv-TasNet) on the GAP9 ultra-low-power MCU achieves a 2 ms inference time (33% of the real-time constraint of 6.25 ms), along with a 108.9× reduction in MAC operations and a 27.7× cycle reduction. Using four timesteps increases the MAC/Cycle ratio to 3.3 while maintaining an inference time of 4.3 ms, less than 18% of the extended real-time budget (25 ms). Combining INT8-BF16 mixed-precision quantization and multi-timestep processing delivers a 4× memory saving at the same performance.

1.5 Thesis Structure

Fig 1.2 summarizes the structure and logic of the dissertation. Following the Introduction, Chapter 2 surveys the state of the art, positioning this thesis within research on low-precision computing, edge platforms, and sequence models. Chapter 3 provides the necessary architectural and methodological background—covering the PULP/GAP9 platforms and the principles of transprecision computing—that underpins the subsequent contributions. Chapter 4 presents a systematic exploration of FP8 and related small-floating-point formats on MCU-class hardware, analyzing accuracy–efficiency trade-offs. Building on these findings, Chapter 5 introduces TransLib, a transprecision kernel library that operationalizes the explored techniques for embedded multi-cores. The dissertation then turns to application and validation: Chapter 6 develops an SOM case study for on-device genomics to quantify real-world gains; Chapter 7 proposes StreamEase, an automated framework that converts non-streaming TCNs into efficient streaming models for real-time inference on low-power MCUs. Chapter 8 concludes by synthesizing insights, discussing limitations, and outlining avenues for future research.

CHAPTER

2

RELATED WORK

Edge AI on MCU-class platforms lies at the intersection of three threads: (i) approximate and transprecision computing, which trades precision for energy and memory, (ii) architectural and ISA support, together with software stacks, which makes low-precision and vectorized kernels practical on RISC-V and Arm MCUs, and (iii) streaming sequence modeling—from Recurrent Neural Networks (RNNs) and Long Short-Term Memory Networks (LSTMs) to Transformers and TCNs— where latency, state size, and data movement dominate costs. This chapter briefly reviews representative work in each area, emphasizing what is most relevant to multi-core near-threshold platforms such as PULP/GAP9.

We first summarize approximate and transprecision principles on resource-constrained MCUs, then outline low-precision number formats (smallFloats and microscaling) and the ISA/hardware features that enable them in practice. Next, we survey platform and library support that turns these ideas into deployable kernels. Finally, we connect to time-series models at the edge, focusing on the advantages and challenges of TCNs for streamability.

2.1 Approximate and Transprecision Computing on MCUs

Because of their flexibility, low power, and low cost, embedded MCUs are the dominant IoT compute platforms, but tight energy and memory budgets complicate the deployment of state-of-the-art ML models. Approximate computing (AxC) exploited the error resiliency of many workloads—including image/signal processing, data mining, robotics, and speech—to trade small quality losses for large efficiency gains [35, 63]. Building on AxC, transprecision computing proposes fine-grained control of precision across the stack (algorithms, software, ISA, microarchitecture) to minimize energy while meeting accuracy targets [69, 71]. Projects such as OPRECOMP established the transprecision agenda and demonstrated system-level methodologies from sub-mW IoT nodes to MW-scale systems [22, 71]. Classical mixed-precision algorithms in linear algebra [37] and signal processing [114] inspired hardware–software co-design with reduced-precision formats and software tooling (e.g., FlexFloat) for design-space exploration [123]. In the IoT setting, the key idea is to dynamically allocate precision where it

matters, keeping memory footprints and compute energy small without compromising application-level quality.

Adopting smallFloats has been highly beneficial for the success of transprecision computing [121]. FP16 and BF16/FP16ALT are widely used to accelerate training and inference while preserving accuracy in many DL workloads; they are now common in near-sensor computing pipelines and MCU toolchains that support half-precision storage and arithmetic. On Google Cloud TPUs (v2/v3+), matrix multiplies run in bfloat16 with FP32 accumulation by default, enabling mixed-precision training that typically matches FP32 accuracy while improving time-to-train and memory footprint [31, 47, 85, 133]. Increasingly, FP8 formats are being investigated for both training and inference [25, 49, 57, 91]. Micikevicius [78] introduced two 8-bit floating-point encodings—E4M3 and E5M2, where E and M denote the number of exponent and mantissa bits, respectively—and showed that FP8 can match FP16/BF16 quality on a broad range of Convolutional Neural Network (CNN)/RNN/Transformer tasks when combined with appropriate scaling and mixed-precision recipes [78]. Comparative analyses highlight when FP8's non-uniform sampling beats INT8 quantization (especially for activations with large dynamic range), and when INT8 remains competitive for fixed-weight inference [130]. On the software side, NVIDIA's Transformer Engine provides FP8 recipes (HYBRID: E4M3 forward, E5M2 backward) with delayed amax scaling for Hopper/Ada/Blackwell GPUs [87]. Intel's FP8 emulation toolkit enabled FP8 exploration on CPU/GPU using PyTorch bindings [41, 58]. Beyond FP8, per-block microscaling data formats mix narrow FP/INT element types with a scale per block, offering attractive accuracy–efficiency trade-offs [102]. For ultra-low-power MCUs, FP8 is not yet a mainstream hardware feature, but transprecision-oriented designs and toolflows strongly motivate storage in FP8/smallFloats and compute in higher precision where needed, to reduce memory traffic—the first-order energy driver on MCUs.

2.2 ISA Extensions for Transprecision

Architectural support is critical to realize transprecision benefits. On RISC-V, Tagliavini proposed smallFloat ISA extensions that add scalar and SIMD operations for mini-float types and IEEE binary16, demonstrating speedups up to $1.9\times$ for scalar and $3.6\times$ for FP8-style vectorized code on ultra-low-power cores [120]. Related open hardware, such as FPnew, adds parameterizable FP units with transprecision formats [70]. On Arm, Helium (M-Profile Vector Extension, MVE) brings 128-bit SIMD with FP16/FP32 and int8/16/32 data types to Cortex-M55/M85, yielding large uplifts in DSP/ML throughput; Arm reports up to $5\times$ DSP and $15\times$ ML speedups vs. prior Cortex-M generations, and even larger gains when pairing Cortex-M55 with the Ethos-U55 microNPU [5, 6, 8]. Helium is now widely supported in compilers and CMSIS libraries, enabling vectorized FP16/int8 kernels that directly benefit TinyML workloads [7].

Near-threshold parallel platforms such as PULP demonstrate that multi-core MCU-class clusters deliver high energy efficiency at a few mW by exploiting parallelism and data movement optimizations [94, 100]. On PULP derivatives like GAP8 and GAP9 (commercial) and research SoCs, software stacks such as PULP-NN accelerate 8-bit and sub-byte QNNs, achieving up to 15.5 MACs/cycle and strong speedups/energy gains over single-core MCUs [27]. Recent transprecision clusters integrate low-power FP units and memory hierarchies to efficiently run mixed-precision workloads in near-sensor settings [82]. GreenWaves' GAP9 advances this trajectory with substantially improved energy per GOps at the core level, targeting battery-powered embedded AI at the edge.

2.3 From Transprecision on MCUs to Edge AI and Streaming TCNs

In recent years, deep learning techniques have achieved state-of-the-art results across a wide range of machine learning tasks [54] [59], driven by groundbreaking advancements in fields such as speech enhancement [125] [75], natural language processing [55], and video object detection [45]. The diverse nature of time-series problems across various domains has led to the development of numerous neural network architectures tailored to address specific challenges in sequential data modeling [59]. Traditional approaches rely on static rule-based systems or signal-processing pipelines processing data sequentially [64]. However, the increasing complexity and variability of real-time data streams have led to the adoption of ML and artificial neural network (ANN) methods. In recent years, Deep Neural Networks (DNNs) have become predominant [81], and designers have proposed several architectures: RNNs [81], LSTMs [38], and TCNs [60].

RNNs have been a foundational architecture for processing sequential data due to their ability to maintain a hidden state. This feature allows RNNs to capture temporal dependencies effectively, making them suitable for tasks involving sequential or time-series data [81]. While RNNs are effective at processing sequential data one timestep at a time, making them suitable for real-time applications, they are challenging to train and suffer from limitations in parallelization during inference [89]. Additionally, their capability to handle long-term dependencies is limited by issues such as the vanishing gradient problem [43] [105] [66]. To address some of these limitations, LSTMs were introduced as an extension of RNNs [38]. LSTMs incorporate memory cells and gating mechanisms, which mitigate the vanishing gradient problem and enable the modeling of longer-term dependencies [81]. These features make LSTMs particularly effective for real-time applications, such as speech recognition and language modeling. Despite their advantages, LSTMs are computationally intensive, and training and inference do not take advantage of the most common parallelization techniques [12] [105]. These limitations have driven the development of alternative architectures, such as Attention-based Transformers and Temporal Convolutional Networks (TCNs) [9].

Transformers are a more recent architecture and overcome several limitations of RNNs and LSTMs by processing sequences using the self-attention mechanisms [132] [12]. This approach allows Transformers to capture long-range dependencies more effectively and efficiently. However, their reliance on self-attention results in high computational demands and memory usage, making them less suitable for resource-constrained environments [56] [106]. Recent advancements in sequence modeling have also introduced Extended Long Short-Term Memory (xLSTM) [12] and State Space Models (SSMs) [32] [33] as alternatives to address the quadratic complexity of self-attention and the slow training of RNNs [90]. However, despite their advancements, both xLSTM and SSMs face limitations, such as high computational complexity and challenges in parallelization, which can hinder their deployment on resource-constrained embedded systems [12].

TCNs leverage dilated convolutions to capture long-term temporal dependencies [10]. Unlike RNNs and LSTMs, TCNs process sequences in parallel, enabling faster training and inference [95]. Moreover, they typically require fewer parameters and provide stable gradients during training [43]. A TCN employs dilated 1-D convolutions to create a temporal receptive field, including the input timesteps contributing to the output at a given point [60]. TCN-based models deliver outstanding performance in several tasks [68] [67]. Recent studies have demonstrated that TCN-based models deliver outstanding performance in several tasks [68] [61] [50] [67]. The experimental findings in [10] show that TCN models achieve ac-

curacy comparable to traditional recurrent architectures such as LSTMs and GRUs in sequence modeling. Furthermore, TCNs offer several computational benefits, including enhanced data reuse [96], flexible receptive field sizes, stable gradients, variable-length inputs, and greater arithmetic intensity [10] [95]. These characteristics make TCNs particularly well-suited for streaming applications, where variable-length inputs and low latency are critical.

Advanced normalization techniques [113] [16] further improved the accuracy of TCN-based models. Conv-TasNet [67] employs global layer normalization (gLN) and cumulative layer normalization (cLN) to normalize inputs across the entire sequence. Despite these strengths, TCNs face significant challenges when deployed in real-time environments, especially on resource-constrained devices, due to substantial computational demands [127] [43]. Despite significant performance advantages across various tasks, the high computational complexity of these methods can limit their broad adoption. In particular, many of these algorithms are unsuited for deployment on embedded systems or in environments with strictly limited computational resources [127].

A notable solution to this challenge is the RT-TCN [48], which minimizes computational and memory demands by reusing results from previous convolution operations, thereby optimizing the network's real-time performance. A framework for keyword spotting models (KWS) that uses a streaming approach is presented in [104]. However, it does not directly address the state reuse and buffering mechanisms required for efficient streaming inference in TCNs. The SOMA hardware accelerator, presented in [29], is designed to support streaming TCNs for KWS applications. While SOMA is optimized for KWS tasks, its specialized hardware limits its flexibility for broader applications beyond keyword detection. A hardware accelerator for TCNs on an FPGA is proposed in [15], designed to efficiently handle varying dilation and stride values. The approach leverages registers to store intermediate results for each layer. However, a limitation of this method is that the entire register is passed to each layer during processing of new inputs, which can lead to inefficient memory bandwidth utilization and increased latency, particularly in resource-constrained environments.

The second part of this thesis addresses the key challenges faced by previous TCN-based methods, particularly the high computational demands and memory constraints that have limited their deployment in real-time applications on resource-constrained devices. The multi-timestep streaming approach significantly reduces the frequency of model inference calls, allowing the system to reallocate computational resources without compromising the real-time performance.

CHAPTER

3

BACKGROUND

Edge inference must deliver real-time performance under power budgets of only a few milliwatts and memory budgets ranging from kilobytes to a few megabytes. Meeting these constraints requires platforms that pair efficient parallelism with fast on-chip memory, and programming models that minimize data movement while exploiting numeric formats beyond full precision. This chapter provides the architectural and methodological background used throughout the thesis.

We first introduce the PULP family of RISC-V systems-on-chip and the commercial GAP9 MCU derived from it. Next, we define transprecision computing—the selective use of different numeric precisions across stages of a program to reduce energy and latency while preserving task quality. Finally, to ground later chapters, we characterize the power and energy behavior of PULP under representative kernels. We quantify the cost of floating-point unit (FPU) sharing and the gains from vectorization and multi-core parallelism, providing concrete guidance for the design choices made in the remainder of the thesis.

3.1 PULP

Edge AI nodes require high throughput and energy efficiency within a power limit of a few milliwatts. PULP [28] is an open-source system-on-chip (SoC) designed for near-threshold parallel computing. It integrates an MCU and a programmable multi-core cluster, together with an on-chip SRAM (L2) that stores resident code and application data. The multi-core cluster consists of a variable number of identical RISC-V cores sharing a Tightly Coupled Data Memory (TCDM) organized into multiple banks. This design allows the cores to access data in a single clock cycle through a low-latency logarithmic interconnect. The software manages data transfers between the L2 and the TCDM using a dedicated Direct Memory Access (DMA) controller. This controller orchestrates data transfers in parallel with core computations, thereby reducing the impact of memory access latency and improving overall system performance. In addition, a third memory level is available in the system, namely an off-chip L3 memory with virtually unlimited capacity.

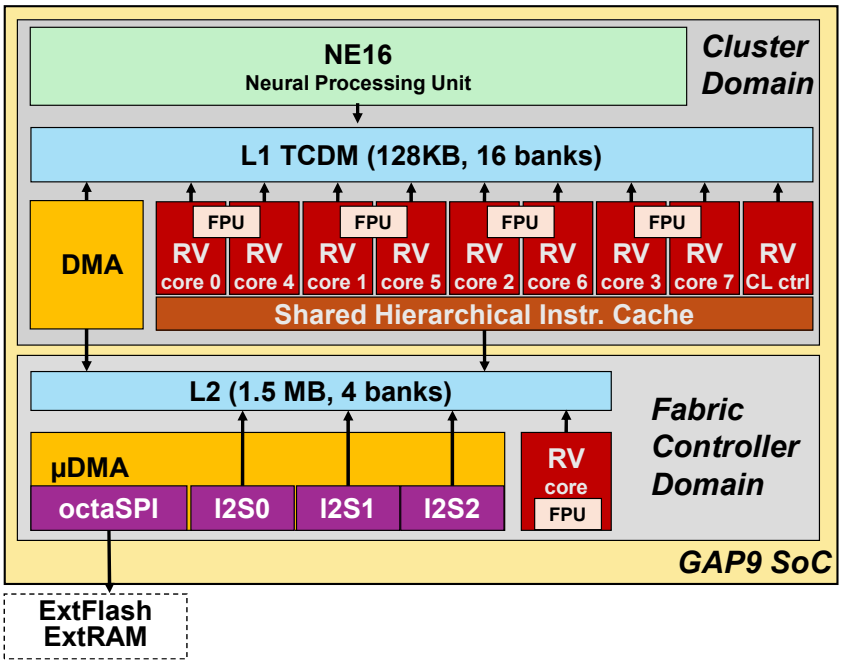


Figure 3.1: GAP9 architecture. RV denotes RISC-V processor cores; FPU denotes shared floating-point units; L1 TCDM and L2 denote on-chip memory levels.

The cluster cores share four FPUs that support FP32, FP16, BF16, and FP8 arithmetic, including cast operations between different formats and cast-and-pack instructions to improve the effectiveness of packed SIMD vector operations [70]. These FPUs are connected via a low-latency interconnect that handles access contention in hardware, allowing multiple cores to share a single FPU seamlessly from a software perspective.

[82] proposes an architecture for a transprecision cluster with a dedicated interconnect that supports shared FPUs for SmallFloat vectorization. The PULP SDK [1] provides a complete software environment, which includes a compiler toolchain, a virtual platform, and the PULP-OS environment. The compiler frontend has been extended to support SmallFloat types. The virtual platform, called GVSoC [14], is an open-source simulator of an entire platform, including core pipelines and memory hierarchy. GVSoC provides a good trade-off between simulation speed, timing accuracy, and completeness. Finally, PULP-OS is a lightweight operating system providing a low-level runtime library to enable parallel execution on the cluster.

3.2 GAP9

GAP9 is an energy-efficient MCU based on the RISC-V architecture, developed by GreenWaves Technologies. It builds upon the open-source Vega SoC architecture [98]. As depicted in Fig. 3.1, the platform comprises two primary domains: the Fabric Controller (FC) and the Cluster (CL). The FC serves as the central control unit, orchestrating thread execution and managing data exchanges with peripheral devices. In contrast, the CL serves as a general-purpose parallel-processing accelerator, featuring nine RISC-V cores enhanced with ISA extensions for AI and DSP. Additionally, the CL integrates four floating-point units. GAP9 supports a supply voltage range from 0.65 V to 0.8 V, enabling maximum operating frequencies of 240 MHz and 370 MHz, respectively. The 0.65 V configuration represents the optimal energy-efficiency point, whereas the 0.8 V setting delivers peak performance.

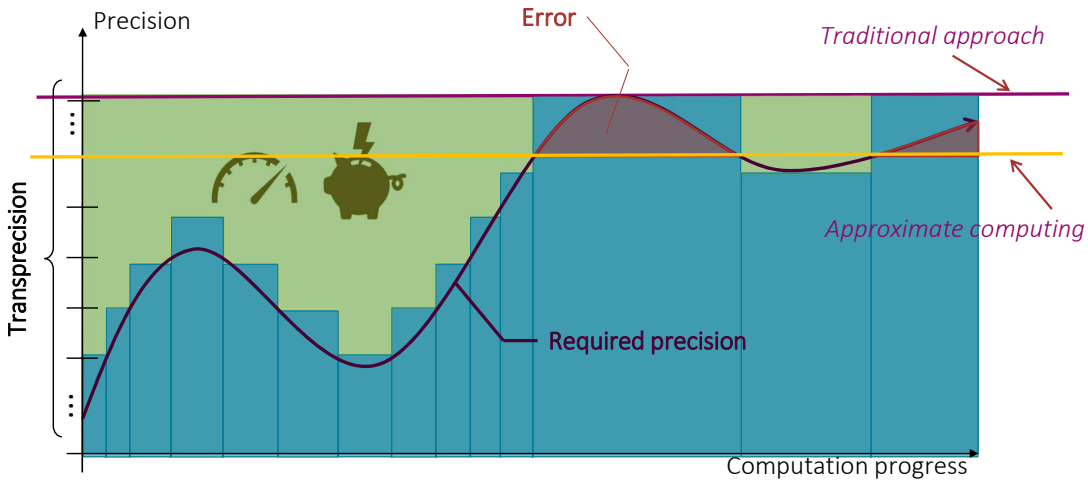


Figure 3.2: A comparison of computation methodologies

3.3 Transprecision

Transprecision computing is an approach for improving efficiency by selecting the numeric precision used in different parts of a program. As depicted in Figure 3.2, in conventional implementations, computations often use a single, high-precision floating-point format throughout, which can increase energy consumption and execution time. Since programs typically consist of multiple stages with different accuracy and dynamic-range requirements, using the same precision everywhere is often unnecessary. Approximate computing (AxC) addresses this by allowing controlled inexactness, often by uniformly reducing precision, but this can introduce avoidable errors in stages that require higher precision.

Transprecision instead orchestrates multiple precisions across the stack (algorithm, software, ISA, and microarchitecture), assigning higher precision only where required to meet application-level quality targets. This enables reductions in compute and memory cost while preserving numerical behavior in critical stages, which is particularly relevant for energy-constrained domains such as IoT sensing, robotics, and embedded analytics.

3.4 Power and Energy Characterization of PULP

As discussed in Section 3.1, the PULP cluster includes four shared FPUs. The FPUs are statically mapped to specific cores in a partial interconnect configuration. Specifically, units 0, 1, 2, and 3 are shared among cores 0 and 4, 1 and 5, 2 and 6, and 3, 7, and 8, respectively. This design choice limits the flexibility of FPU sharing across cores but simplifies the interconnect towards the critical path of the cluster. As a result, it guarantees high compute efficiency by maintaining a single-cycle latency behaviour for FP instructions. We analyzed the effects of architectural design choices on the energy and power consumption of the GAP9 platform, considering the cores, memory, and interconnects. Table 3.1 presents the outcomes of this analysis, executing a compute-intensive kernel (i.e., a matrix multiplication) on different configurations.

Sharing an FPU between two cores results in a $1.03\times$ and $1.02\times$ increase in cycles and energy consumption compared to the non-shared scenario. These values are roughly equivalent when comparing the power consumption of code using FP instructions to that of code using integer operations on a single core. However, when utilizing 8 cores, FP32 exhibits increased cycles, energy consumption, and average power by $1.09\times$, $1.25\times$, and $1.22\times$, respectively.

Table 3.1: Normalized cycles, energy, and average power for FP32, INT32, and vectorized FP16 on the PULP SoC.

	Cores	Baseline	Cycles	Energy	Average Power
Shared FPU	2	Non Shared FPU	1.03x	1.02x	0.99x
FP32	1	INT32	1.00x	1.00x	1.06x
FP32	8	INT32	1.09x	1.25x	1.22x
Vectorized FP16	1	FP32	0.56x	0.46x	1x
INT32	8	INT32, 1 core	0.13x	0.19x	1.44x
FP32	8 (7 idle)	FP32, 1 core	0.99x	0.99x	1x

This result is due to the FPU's higher circuit complexity.

Moreover, we evaluated the impact of vectorization and parallelization. FP16 vectorization reduces the cycles and energy consumption by $0.56\times$ and $0.46\times$, respectively. Vectorization achieves this by minimizing the load and store operations, thereby reducing the energy expended during a load-execute-store process for each set of input operands. However, adopting vectorization in real applications incurs additional overhead due to conversions from other formats or parts of inherently scalar algorithms, preventing an ideal reduction in cycles (by $0.5\times$). To examine the impact of parallelization without interference from FPU sharing, we compared 8-core and 1-core performance for INT32 computations. The results reveal that employing 8 cores reduces the number of cycles and energy consumption by $0.13\times$ and $0.19\times$, respectively, compared to using only 1 core. Finally, to evaluate power-saving policies when cores are idle, we ran the benchmark on 8 cores, assigning the entire computation to a single core while the others were clock-gated. The results are similar in the two cases, indicating that clock gating is effective when cores are idle.

CHAPTER

4

ADOPTING FP8 FORMATS OUTSIDE THE DEEP-LEARNING DOMAIN

Recently, there has been increasing focus on using FP8 formats for training and inference of deep neural networks [86, 92]. In this context, we position our study alongside and beyond these FP8 efforts. In [52], the authors explore various choices of mantissa and exponent bit widths and analyze their impact on performance with different data and weight distributions[1]. Their main finding suggests that choosing an appropriate number of exponent bits can yield higher accuracy with the FP8 format than with INT8. However, their analysis focuses solely on model accuracy, ignoring hardware factors such as power consumption and latency. Consequently, we turn to a broader, system-aware perspective. In this section, we explore the benefits of utilizing the FP8 format beyond deep learning applications. We target an optimized implementation for the PULP platform based on the RISC-V ISA. Our primary objective is to assess how adopting FP8 affects critical metrics, including power consumption, execution time, and memory footprint.

4.1 Methodology

4.1.1 High-level simulations

A floating-point number can be represented as $f = (-1)^s 2^{(e-bias)}(1.m)$, where s is the sign, e (exponent) determines the dynamic range, and m (mantissa) controls the precision. Given a fixed bit-width, increasing the exponent bitwidth expands the dynamic range at the expense of reduced precision. One of our objectives is to explore FP8 data formats, considering different exponent and mantissa bit-width combinations (ExMy). We adopted these formats in matrix multiplication (Matmul) and convolution (CONV) kernels. We first focused on the kernels with random normal data distributions characterized by defined standard deviations (σ) and means

[1] https://github.com/Qualcomm-AI-research/FP8-quantization

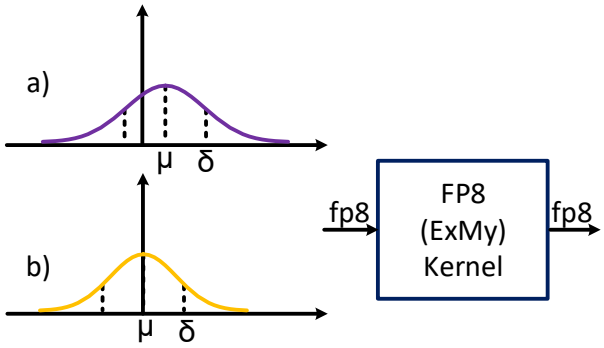


Figure 4.1: Analysis of a) random data distribution, b) scaled random data distribution for FP8 kernels.

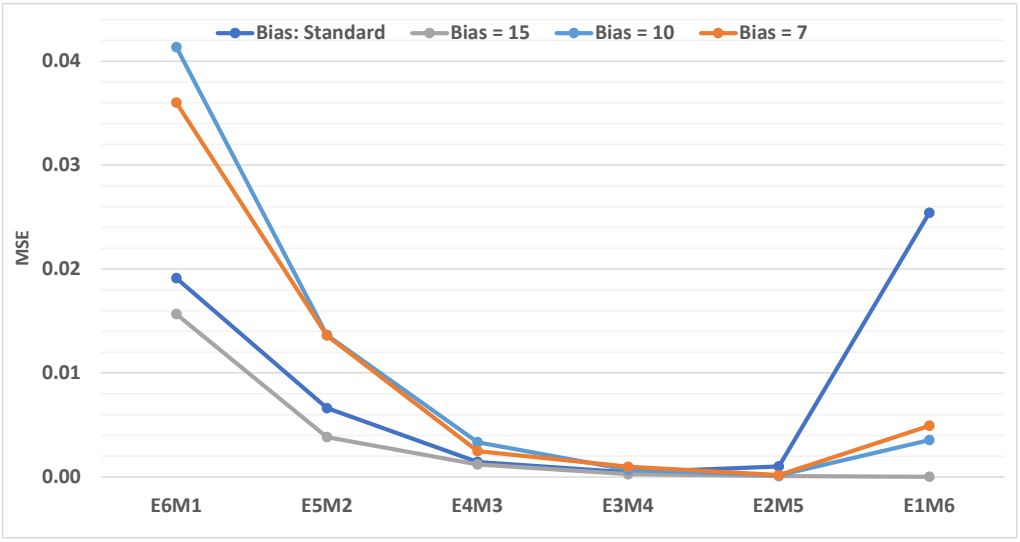


Figure 4.2: Effect of varying biases in FP8 formats on MSE for the CONV kernel at a standard deviation of 0.5.

(μ) (Figure 4.1). Given the constraints imposed by dynamic ranges and precision levels across different FP8 formats [52], ensuring that the input data distribution aligns with the dynamic range of a specific FP8 format is vital to prevent overflow and underflow during computations.

The FP representations are non-uniform; they offer greater accuracy for values near zero but reduced precision for greater values [130]. As a result, distributions with higher means will be represented less precisely, leading to higher error metrics in the corresponding computations. To address this issue, we considered two approaches. The first approach mirrors the methodology employed in other FP formats, shifting and scaling the distribution to match the suitable range of the targeted FP8 format [110].

The second approach, inspired by techniques explored in deep neural networks [86], involves modifying the bias within floating-point formats. This approach is similar to static scaling. However, as demonstrated in [52], it allows for representing a greater volume of data near zero using the non-uniform number representation characteristic of FP formats.

4.1.2 Software library for embedded targets

In this study, we aim to explore further the applications of FP8 data formats in practical hardware environments. Unlike previous research, which focused primarily on model accuracy

Table 4.1: Evaluation of error metrics (MSE, RAE, and accuracy) for MATMUL and CONV kernels in FP16 and the best FP8 configurations for normal distributions with defined mean and standard deviation.

	FP16		FP8			Data Distribution	
Kernel	MSE	RAE	FORMAT	MSE	RAE	Mean	STD
Matmul	1.58e-06	7.81e-04	E3M4	1.16e-01	4.97e-02	0	1
	1.58e-06	7.81e-04	E3M4	7.55e-03	5.45e-02	0	0.5
	2.37e-13	7.83e-04	E5M2	1.39e-08	2.01e-01	0	0.01
CONV	4.14e-06	5.14e-04	E3M4	2.37e-02	3.55e-02	0	1
	1.83e-07	5.04e-04	E3M4	8.76e-04	3.66e-02	0	0.5
	1.27e-14	4.66e-04	E5M2	1.78e-09	1.46e-01	0	0.01

without considering power consumption and latency impacts specific to hardware, our study delves deeper into these hardware aspects. We thoroughly analyze the trade-offs of using FP8, accounting for its impact on execution time, energy consumption, and memory utilization. We have developed and implemented a library of C code optimized for embedded targets. The kernels in the library can execute in both scalar and vector modes. Compiler toolchains like GCC and Clang support vector extensions to the C standard, enabling SIMD-like sub-word parallelism. Furthermore, we experiment with various parameters in our plain C code implementations by incorporating preprocessor directives that enable multi-core processing and performance monitoring using the PMSIS interface available for PULP platforms[2].

To examine the impact of various FP data types on energy and power consumption, we conducted a case study using the GAP9 platform.

4.2 Experimental Results

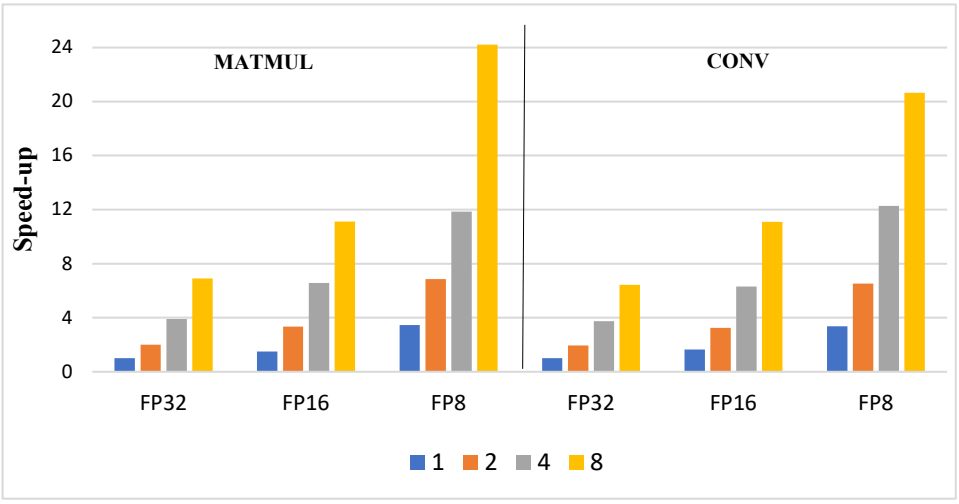


Figure 4.3: (Speedup of parallel (1, 2, 4, 8 cores), vectorized (FP16/bfloat16, FP8) using 1, 2, 4, and 8 cores relative to the FP32 (1 core) baseline.

[2] https://github.com/pulp-platform/pulp-sdk

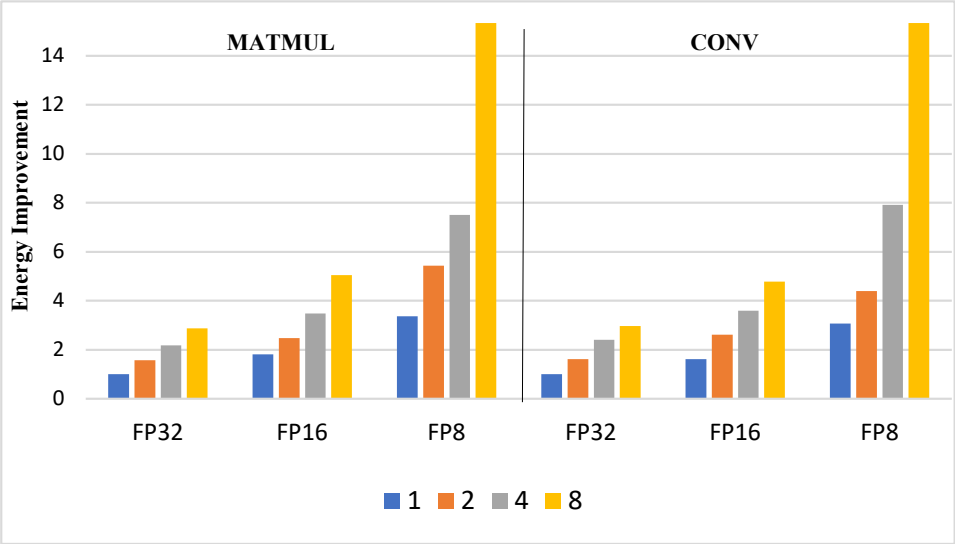


Figure 4.4: Energy improvement of parallel (1, 2, 4, 8 cores), vectorized (FP16/bfloat16, FP8) using 1, 2, 4, and 8 cores relative to the FP32 (1 core) baseline.

4.2.1 High-level simulations

We explored different FP8 formats (ExMy) and FP16 for kernels in Table 4.1. Since the FP16 and bfloat16 results were very similar, we only reported the FP16 results. To facilitate the experimental setup, we employed an FP simulator[3], as detailed in [52]. To analyze the impact of FP8 on the output, we used the error metrics described in [84] and compared them to FP32. As shown in Table 4.1, Mean Squared Error (MSE) and Relative Absolute Error (RAE) are the main metrics for our evaluation.

Table 4.1 illustrates how the data distribution affects the best FP8 format for each kernel. When the data distribution is centered around zero ($std = 0.01$), E5M2 outperforms other FP8 formats, achieving the lowest MSE values. This is because E5M2 can cover the widest range and more accurately represent smaller values close to zero. However, accumulation and multiplication operations require higher precision and a broader dynamic range as the standard deviation increases. Thus, for MATMUL and CONV kernels with $std = 0.5$ and $std = 1$, E3M4 is the best FP8 format in terms of MSE and RAE metrics. Error metrics for FP8 representations are greater than for FP16. However, the numerical results for the kernels are still within acceptable limits.

As explained in Section 4.1.1, the distribution of input data significantly influences the error metrics of FP8 kernels. To address this, we explored two approaches (i.e., scaling and adjusting the bias). We evaluated the MATMUL kernel using FP8 formats with greater standard deviations and means. Through our experiments, we observed that by scaling the input data distributions and shifting them using $\alpha = \frac{desired\ \sigma}{input\_data\ \sigma}$ and μ respectively, we achieved the same results as those presented in Table 4.1. Here, the variable $desired\ \sigma$ represents the standard deviations listed in Table 4.1. Also, our experiments aimed to assess the impact of adjusting the bias in FP8 representation.

Figure 4.2 shows how these bias alterations affect the MSE for the CONV kernel. As expected, increasing the bias for FP8 formats with more mantissa bits expands the dynamic range. Consequently, the MSE decreases when the bias exceeds the standard value used by IEEE FP formats ($2^{(exponent\ bit\ widths\ -1)} - 1$). For example, with a bias value of 15, E1M6

[3] https://github.com/Qualcomm-AI-research/FP8-quantization

exhibited the lowest MSE. This is due to the increased bias and six-bit mantissa of E1M6, which provides sufficient precision to accurately handle smaller values near zero. However, for non-standard biases below 15 (e.g., 7 and 10), E2M5 had a lower MSE than E1M6, primarily because of its broader dynamic range coverage.

4.2.2 FP8 hardware support

We performed a systematic analysis of kernels to investigate the potential benefits of FP8 hardware support:

- Matmul can be easily vectorized with a SIMD approach, making it an ideal choice as the initial test case in exploring the advantages of using FP8.
- CONV exhibits an interesting combination of sequential and parallel executions. While some parts require sequential execution, most can be vectorized and parallelized.

We utilize internal performance counters to measure FP32, FP16, and bfloat16 cycles on the GAP9. For power consumption measurements, we used Nordic Semiconductor's Power Profiler Kit 2 (PPK2) with a sampling frequency of 100 kHz. While GAP9 supports parallelization and vectorization, the native support for the FP8 data type is unavailable, but it is being considered for future iterations of the SoC, and it is already available on the FPNEW hardware unit [4]. Therefore, we measured the latency by adding FP8 support to the event-driven simulator GVSoC[5], which provides cycle-accurate simulations. For energy measurements, we considered a PULP cluster synthesized with Synopsys Design Compiler 2019.12 and LVT libraries from the 22 nm FDX technology from GlobalFoundries [99]. The energy estimation was conducted using Synopsys PrimeTime 2019.12. Value change dump (VCD) traces were extracted from parasitic-annotated post-layout simulations using Mentor Modelsim 2008.06.

Figures 4.3 and 4.4 depict the improvements in execution time and energy saving obtained by varying the numbers of cores (1, 2, 4, and 8) and leveraging both parallelism and vectorization. For all subsequent results, the baseline is FP32 with 1 core. Without vectorization, FP32 yields an average speedup of $1.98\times$, $3.88\times$, and $6.67\times$ on 2 cores, 4 cores, and 8 cores, respectively. The impact of vectorization becomes evident when examining FP8, which enables the concurrent execution of four 8-bit FP operations. When considering the vectorized FP8 configuration, we can observe notable variations in speedup among the kernels. This effect is particularly evident when comparing MATMUL and CONV kernels against FP32 on a single core, where the speedups are $3.47\times$ and $3.36\times$, respectively. As the number of cores increases, we observe a further improvement in the speedup rate when using FP8. In an 8-core configuration, the MATMUL and CONV operations achieve speedup metrics of $24.21\times$ and $20.66\times$, respectively.

The same observation applies to vectorized FP16/bfloat16, involving two simultaneous 16-bit FP operations. Specifically, the MATMUL kernel achieves a $11.1\times$ speedup on an 8-core setup. In terms of energy saving, similar explanations apply. On average, using FP32 results in energy savings of $1.67\times$, $2.5\times$, and $3.5\times$ on 2, 4, and 8 cores, respectively, compared to the baseline. The FP16/bfloat16 vectorization allows for even higher savings with ratios of $1.72\times$, $2.65\times$, $4\times$, and $5.61\times$ on cores from one to eight. Moreover, vectorized FP8 offers significantly greater energy savings at rates $2.97\times$, $5.07\times$, $7.37\times$, and $15.05\times$ versus the baseline. In the case of FP8, the achieved savings are slightly lower than the optimal maximum

[4] https://github.com/openhwgroup/cvfpu
[5] https://github.com/gvsoc/gvsoc

(which is twice that of 16-bit types). This difference can be attributed to the significant impact of pack/unpack operations: on a typical RISC architecture, handling the four operands of an FP8 vector requires two instructions, whereas 16-bit FP types require only a single instruction. Leveraging FP8 can significantly reduce the memory footprint, up to $4\times$ compared to the FP32 format.

4.3 Conclusion

This chapter highlights the crucial role of the FP8 format in optimizing applications beyond the deep-learning context. Our findings demonstrate that it is possible to maintain acceptable accuracy within the domain constraints, improving speed and energy efficiency on ultra-low-power targets. Additionally, its compact nature results in significant memory savings. Combining efficient data formats with advanced hardware platforms points toward a promising future where computational strength, efficiency, and sustainability converge. The future section will involve benchmarking with prototypical applications and providing a more comprehensive evaluation of FP8's performance in real-world scenarios.

CHAPTER

5

TRANSLIB: AN END-TO-END WORKFLOW FOR TRANSPRECISION COMPUTING

This chapter presents TransLib[1], an open-source kernel library for the systematic study of reduced-precision and transprecision computing. TransLib exposes configurable precision settings for 8-bit and 16-bit FP data types and enables evaluating trade-offs between fixed-precision and mixed-precision configurations. We evaluate TransLib on two parallel platforms for IoT end-nodes, PULP-Open and GAP9, each coupling a 32-bit MCU with a parallel compute cluster supporting DSP extensions and multiple FP formats. After that, as a use case, we exercise a suite of DSP/ML kernels (e.g., FIR, convolution, FFT) and instantiate a streaming Pan–Tompkins ECG detector to illustrate applicability.

5.1 TransLib Design

TransLib provides a hardware-aware and precision-configurable workflow for the development and evaluation of DSP/ML kernels on IoT end-nodes (Fig. 5.1). It combines Python generators and reference models with optimized C implementations to enable controlled exploration of fixed- and mixed-precision execution. At the algorithm level, golden models generate inputs and reference outputs while emulating execution semantics, such as fused-MAC operations and SIMD execution, via explicit policy flags. An exploration layer supports systematic sweeps across multiple numerical formats and scaling strategies, and records the resulting accuracy metrics. In parallel, C implementations support scalar and SIMD variants as well as single- and multi-core execution modes. These implementations enable fair and reproducible analyses of the accuracy–performance–energy trade-off. The following subsections describe the principal components of TransLib.

[1] https://github.com/ahmad-mirsalari/TransLib

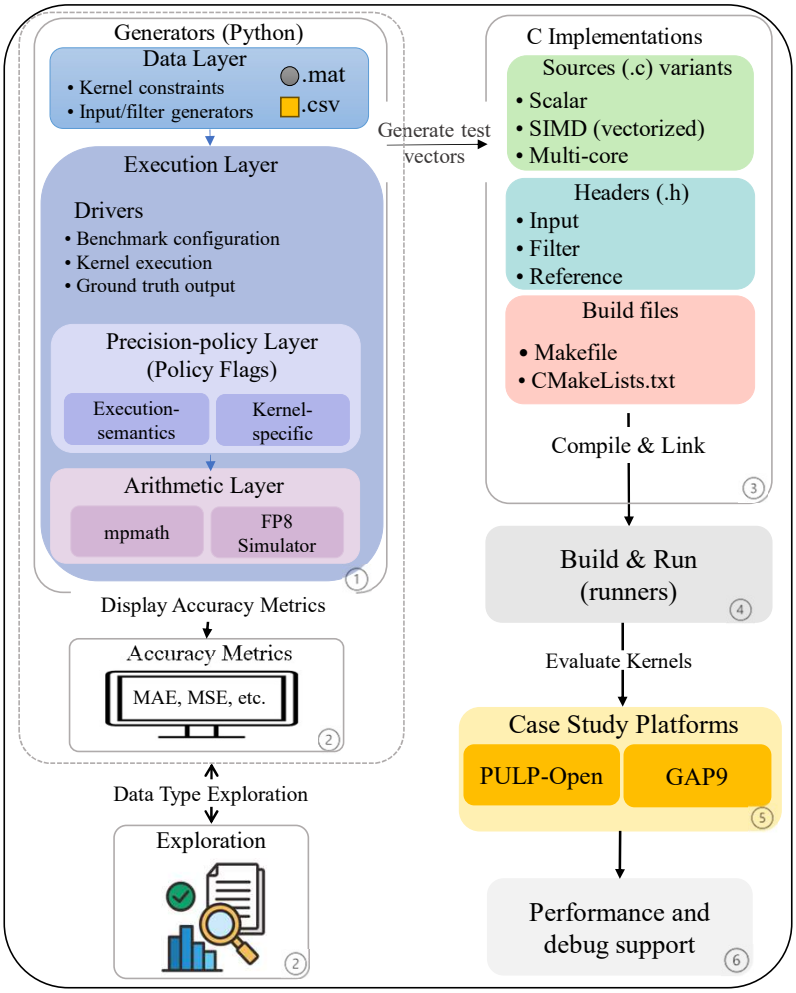


Figure 5.1: Workflow of the TransLib development and evaluation framework.

Table 5.1: Brief description of TransLib kernels.

Kernel	Description
CONV	convolution operation typically used to extract ‘features’ from an input
MATMUL	matrix multiplication, a widely-used linear algebra operation
FIR	finite impulse response filter, digital filter with various applications in data acquisition and analysis
DWT	discrete wavelet transform, standard kernel used for feature extraction
KMEANS	unsupervised learning algorithm, which groups an unlabeled dataset into different clusters
SVM	support vector machine, supervised learning method used for classification, regression, and outliers detection
FFT	fast fourier transform, mathematical method that transforms a signal from the time domain to the frequency domain

5.1.1 TransLib kernels

To enable representative evaluation of transprecision techniques, TransLib includes a set of kernels commonly employed for filtering, feature extraction, classification, and basic linear algebra (Table 5.1). For the Support Vector Machine (SVM), we implemented two variants, namely the Linear and Radial Basis Function (RBF) kernel (also called the Gaussian kernel). For data-driven kernels (e.g., KMEANS/RBF), the generators load `.mat/.csv` datasets—such as the Urban Land Cover [46] and pre-recorded ECG datasets [73]— to generate reproducible test vectors along with golden references for accuracy evaluation, as illustrated in Fig. 5.1. Further details on the kernel configuration options are described in Section 5.1.2.

5.1.2 Generators

To accurately evaluate fixed- and mixed-precision kernels on parallel and SIMD-enabled embedded platforms, a golden reference must capture not only arithmetic precision but also execution semantics such as rounding behavior, reduction order, and parallel accumulation. A single, unified reference model is insufficient because it tightly couples data generation, execution semantics, and precision policies, making it difficult to isolate sources of numerical error or adapt the model to different hardware targets. To address this challenge, we structure the golden reference as four cooperating layers, each explicitly designed to separate concerns that directly impact numerical accuracy, reproducibility, and usability.

5.1.2.1 Data layer

This layer ensures reproducible and representative evaluation by decoupling dataset generation from execution semantics. Recorded datasets (e.g., `.csv`, `.mat`) are loaded, and synthetic test cases are generated while enforcing kernel-specific constraints (e.g., FFT sizes must be powers of two). All inputs are represented as PyTorch tensors, which serve as the data exchange format within the generators.

5.1.2.2 Execution layer

This layer orchestrates the execution of each kernel by coordinating kernel configuration, precision-policy selection, and arithmetic backends. Kernel-specific Python drivers implement the reference execution pipeline. For each benchmark configuration, they (i) instantiate the kernel parameters (problem sizes, boundary conditions, tensor layouts, and parallelization settings), (ii) use the configured precision-policy and arithmetic layers to execute the kernel at the configured precision T_{out}, (iii) collect the resulting tensors as golden-reference artefacts for validating the C implementations.

5.1.2.3 Precision-policy layer (what/when)

This layer specifies what arithmetic semantics to apply and when rounding may occur, addressing a key limitation of conventional reference models. The model supports fixed-precision and mixed-precision via explicit policy flags (e.g., MAC, vectorization, casts). These policies are grouped into (i) execution-semantics flags, which control rounding and reduction behavior, and (ii) kernel-specific flags, which reflect algorithm-dependent precision choices. Table 5.2 summarizes the available flags, their numerical effects, and the scenarios in which they should be applied.

Table 5.2: Flags used in the golden reference model and optimized C implementations.

Flag	Category	Effect
MAC	Execution semantics	Enables fused multiply–accumulate
VECTOR	Execution semantics	Enables SIMD-style reduction
MFAUX	Execution semantics	Emulates auxiliary high-precision paths
CAST	Execution semantics	Forces explicit casts between FP formats
TRANSPOSE	Kernel-specific	Enables matrix pre-transposition for vector loads
REVERSED	Kernel-specific	Controls FIR tap order to match vector access
CORES	Kernel-specific	Enables multi-core execution and parallel reduction
STATS	Runtime (C code)	Collects performance and energy statistics
DEBUG	Runtime (C code)	Enables verbose logging and intermediate output
FABRIC	Runtime (C code)	Enables fabric-controller execution
CORES	Runtime (C code)	Selects the number of active compute cores

Execution-semantics flags These flags emulate instruction-level behaviors that affect rounding, reduction order, and thus intermediate precision. In doing so, the golden model captures arithmetic effects common across a wide range of DSP/ML ISA extensions.

- *MAC flag*— Emulates the fused multiply-and-accumulate (FMA/MAC) as implemented in many DSP ISAs. Following IEEE-754 single-rounding semantics, the product is computed at extended precision and rounded once after accumulation. This modeling approach eliminates the rounding errors that occur when multiplication and accumulation are emulated as separate operations.

- *Vector flag*— Models SIMD patterns in which elements are multiplied in parallel and then summed via a tree reduction. This order differs from left-to-right scalar accumulation and affects rounding. Since floating-point arithmetic is neither associative nor commutative, this modified reduction tree produces distinct rounding behaviors; the flag isolates and analyzes these effects.

- Cast flag— Emulates the effects of explicit type casts between FP formats. Users can enforce casting at specific points (e.g., performing low-precision multiplication followed by higher-precision accumulation).

Kernel-specific flags Some kernels expose additional design flags that reflect algorithm-specific implementation choices. These flags are summarized in Table 5.2 and include options such as matrix pre-transposition (MATMUL), reversed tap order (FIR), and multi-core reduction strategies (KMEANS).

5.1.2.4 Arithmetic layer (how)

Given the selected precision policy, the arithmetic layer executes each operation with explicitly controlled rounding, ensuring that intermediate results match the behavior of the target hardware. We augment PyTorch to enforce single-rounding behavior—each operation is rounded exactly once to the target format at the policy-approved point—avoiding double

rounding from implicit widen/narrow paths. We use `mpmath` [83] to emulate fused multiply–add (FMA/MAC) operations and apply rounding directly to the target precision. For FP8 support, we rely on an external simulator[2] that implements the approach described in [53].

5.1.3 Accuracy metrics

While standard, application-agnostic error metrics provide consistent numerical comparisons, they often fail to capture the functional impact of precision-induced errors in real workloads, where error sensitivity can be highly non-uniform across outputs. To consistently quantify precision effects, we report standard, application-agnostic error metrics—Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Relative Absolute Error (RAE) [84]. We extend PyTorch's accuracy utilities to accept `bfloat16` tensors and mixed-precision outputs without unintended up- or down-casts. To address application-specific requirements, the framework supports extensible accuracy metrics.

5.1.4 Fused Operations for Mixed-Precision Support

Supporting mixed-precision arithmetic generally requires promoting low-precision operands to a higher-precision accumulator. These promotions introduce additional operations that increase execution time and may also incur secondary costs, such as higher register pressure and increased memory traffic. As a result, mixed-precision accumulation can negate part of the efficiency gains expected from reduced-precision computation.

Recent embedded architectures have begun to address this issue by introducing fused mixed-precision scalar instructions that implicitly promote operands and perform accumulation within a single operation. The smallFloat ISA extensions, supported by the PULP platforms [121], augment the RISC-V specification with a set of reduced-precision FP formats (assembly suffixes: b for FP8, h/ah for FP16/bfloat16). The specification also introduces mixed-precision arithmetic, supporting 8- and 16-bit operands with FP32 accumulation through the FAUX extension.

In addition to fused scalar promotions, TransLib introduces a novel fused mixed-precision dot-product-and-sum instruction that treats promotion, multiplication, reduction, and accumulation as a single conceptual operation. Embedding these semantics into the golden model enables designers to quantify the potential benefits of such an instruction in terms of accuracy, cycle count, and memory traffic.

TransLib golden models emulate the hardware behavior by providing a hardware-mixed (mfaux) flag (which performs promotions/accumulation as the fused unit would) and a vectorization flag (which applies the same policy in the SIMD path when operand types match). The framework supports two instructions, (i) scalar mixed-precision fusion (`fmacex.s.`x) and (ii) its SIMD counterpart (`vfsdotpex.s.`x) for vectorized dot products.

5.1.4.1 Scalar Mixed-precision Operations

Without native hardware support, multiply–accumulate (MAC) operations explicitly cast both operands to the accumulator type and then perform multiplication and addition. Concretely, when the accumulation type is explicitly specified in the source:

$$\begin{aligned} &\texttt{acc} \mathrel{+}= (T_{\text{acc}})\, a \;*\; (T_{\text{acc}})\, b, \\ &\qquad a, b \in \{\text{FP8}, \text{FP16}, \text{BF16}\}, \quad \texttt{acc} \in T_{\text{acc}} \end{aligned} \tag{5.1}$$

[2] https://github.com/Qualcomm-AI-research/FP8-quantization

If we put this code inside a loop, each iteration promotes a and b to T_{acc} and then performs the multiplication and an addition in T_{acc} precision (5–6 instructions).

Enabling the compiler flag `-mfaux` instructs the backend to detect this mixed-precision accumulation pattern and treat it as a candidate for fusion (our current FAUX design uses $T_{\text{acc}} = \text{FP32}$, but the formulation is general). In this case, the compiler emits a single fused operation `fmacex.s.`x (where `x` denotes the operand-type suffix) that (i) loads 8/16-bit operands, (ii) promotes them internally to T_{acc}, (iii) performs the multiplication, and (iv) accumulates in acc. If the final stored data-type T_{out} is narrower than T_{acc}, a single conversion is applied once after the loop (e.g., `out = (T_out)acc;`).

5.1.4.2 Vectorial Mixed-precision Operations

Without a native fused mixed-precision SIMD instruction, each vector lane must be cast to the accumulator type before the multiplication, and the subsequent reduction requires lane extraction/shuffles. This overhead significantly reduces the benefits of SIMD execution.

The MiniFloat-NN ISA extension [13] proposes a set of expanding operations that support low-precision NN training. To provide similar capabilities while keeping a general-purpose scope, we extend the smallFloat extension with a fused dot-product instruction (vfsdotpex.s.x), defined under two contracts: (1) the result is accumulated in a FP32 register, and (2) both input operands share the same precision. Under these conditions, the compiler lowers the vector MAC pattern to `vfsdotpex.s.`x. The instruction (i) loads vectors of 8/16-bit operands, (ii) promotes elements internally to FP32, (iii) performs pairwise products, (iv) computes a reduction sum, and (v) accumulates the result in FP32. If the final output type in the C code is narrower than FP32, the FP32 accumulator is down-cast once after the loop, eliminating per-iteration conversion overhead. Heterogeneous operand pairs (e.g., `FP8`$\times$`FP16`) are not supported in the current design to preserve a compact datapath and instruction encoding; however, they can be enabled with modest ISA and microarchitectural extensions if required by future workloads.

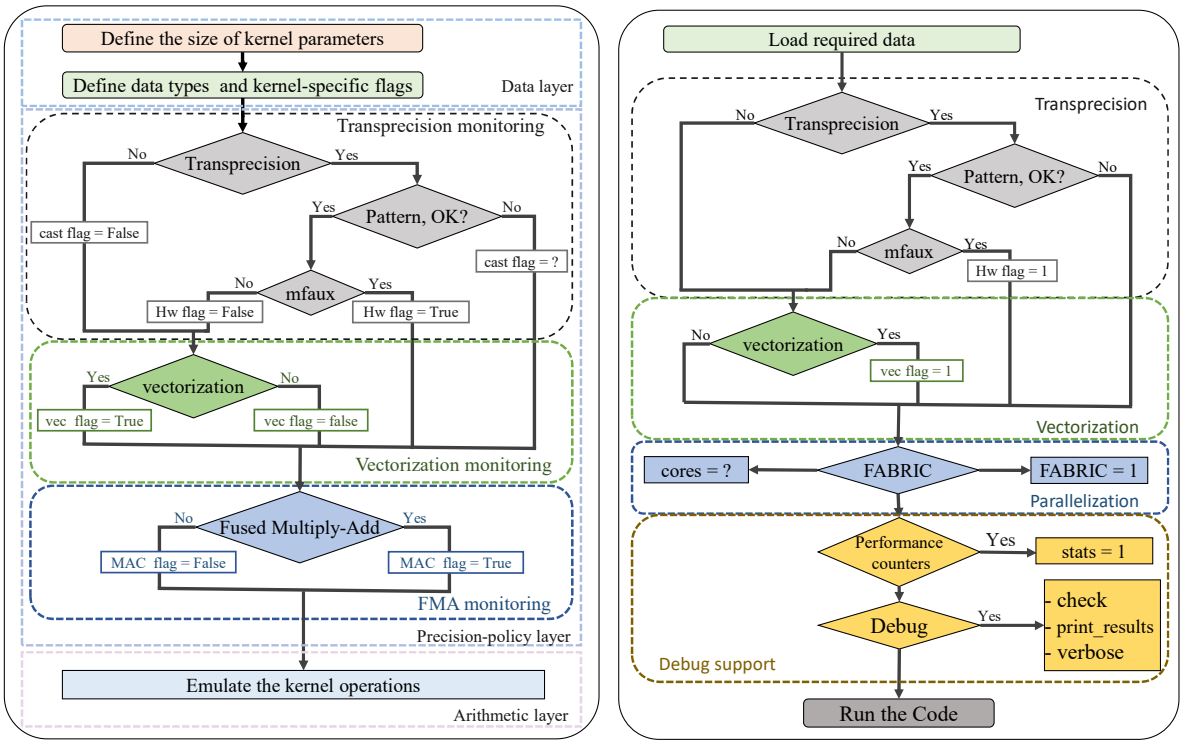
5.1.5 Optimized C code

The TransLib kernels are implemented in C and incorporate a set of optimizations that are portable across MCU-class platforms. These optimizations are designed to support both fixed-precision and mixed-precision execution.

5.1.5.1 Parallelization and vectorization

Each kernel supports multi-core execution through a configurable parallelization parameter, enabling the workload to be partitioned across available processing elements. Parallel execution relies on two abstract primitives provided by the software environment: a mechanism to identify the executing core and a synchronization barrier to coordinate different computation phases.

GCC and Clang compiler toolchains support the vector extension to the C standard, enabling designers to provide kernel variants that use packed-SIMD parallelism. Due to the overhead associated with precise memory alignment and the data-layout transformations, vectorization is not currently supported in FFT and DWT kernels. Since FP8 support is not yet mainstream in the commodity compiler toolchains, we provide an experimental GCC-based toolchain that introduces a `float8` data type at the C level to enable early prototyping and evaluation.



(a) Data and execution layers in generators

(b) C configuration

Figure 5.2: Details of the data and execution layers in the high-level and the C configuration. Hw flag is the hardware mixed promotion flag.

5.1.5.2 Kernel-specific optimizations

To reduce memory traffic and improve SIMD utilization, the CONV kernel reuses horizontally aligned input vectors across successive windows. Rather than sliding the kernel left-to-right, which disrupts vector alignment and forces reloads, we slide it vertically over the input. Using this technique, previously loaded horizontal vectors are valid for the next window and can be reused.

DWT applies a convolution by sliding a filter (corresponding to the wavelet) over the signal. In the baseline layout, the input indices run backward while the wavelet indices run forward, breaking contiguity and thus SIMD on smallFloat formats. We restore contiguity by (i) reversing the inner loop and (ii) storing the wavelet coefficients in reverse order, so both streams advance with the same stride.

In MATMUL, the columns of the second matrix are strided and non-contiguous, which limits SIMD efficiency. We support two vectorization paths: (i) a shuffle-based path that broadcasts first matrix elements across lanes while streaming contiguous rows of the second matrix (portable but with shuffle overheads), and (ii) a pre-transposed path where the second matrix is stored in column-major order to enable aligned vector loads and dot-product reductions. A transpose flag selects the latter.

In the SVM with the RBF kernel, the exponential function plays a central role in computing the Gaussian similarity between input vectors and support vectors. To accelerate this computation, we employ a fast exponential approximation based on the method proposed by [108], offering a significant reduction in computational complexity with minimal impact on accuracy.

5.1.6 Debug support

To facilitate performance analysis and support optimization and regression testing, TransLib offers several debugging features that enable users to inspect outputs at adjustable levels of verbosity via a set of configurable C-level flags. print_results flag prints the kernel output on the standard output. check flag compares the outputs against the reference results produced by the golden model. When deeper inspection is required, the verbose flag enumerates all mismatches. Additionally, a set of per-core hardware performance counters can be activated through the stats flag. Many hardware platforms support this feature, which can report execution statistics such as the number of executed instructions and cycle counts across various execution metrics (e.g., total cycles/instructions, active cycles, shared memory contentions, FPU stalls, load stalls, branch stalls, and instruction cache misses).

5.1.7 Development flow

Figs. 5.2a and 5.2b summarize the flow of data and execution layers in generators, as well as the corresponding C configurations. As a first step (Fig. 5.2a), the user specifies the kernel size and the desired data types. Next, using the flags described in Section 5.1.2, the golden model emulates the hardware and generates the reference outputs. When operating in transprecision mode, the generator inspects the operand types to determine whether they match a supported mixed-precision (scalar or vectorial) pattern (e.g., $\texttt{FP8}\times\texttt{FP8}\rightarrow\texttt{FP32}$). If it finds a match, the user may enable the hardware-mixed flag, which instructs the backend to use fused mixed-precision instructions that perform on-the-fly operand promotion and FP32 accumulation. If the hardware-mixed flag is disabled, the user may instead activate the vectorization flag to generate a SIMD mixed-precision vectorization implementation. When neither fused hardware support nor SIMD vectorization is selected, the system falls back to a cast mode, in which explicit software casts to the target data type are inserted before each multiply–accumulate operation. Once the flag configuration is finalized, the golden model executes the kernel, computes the accuracy metrics, and stores the generated parameters and intermediate results in a file for use by the C implementation.

As a first step, the C implementation loads the input data generated by the Python-based golden models. Kernel-level configuration is controlled through flags specified in the Makefile, which enable the optimization features described in Section 5.1.5. Vectorization (vector flag) is supported for smallFloat data types, while parallel execution can target either a single-core MCU (FABRIC) or a multi-core accelerator, with the number of active cores specified through the (cores) flag. Additional debugging and performance-monitoring capabilities are available through the debug and stats flags, respectively, as outlined in Section 5.1.6. Once all configuration options are set, the C implementation executes the kernel according to the selected execution and precision policies.

5.1.8 Automated Precision-Aware Exploration

To systematically quantify the impact of numerical precision on accuracy across DSP and ML workloads, TransLib embeds an automated precision-aware exploration engine within the golden model infrastructure, as depicted in Fig. 5.1. For each kernel, the framework provides a configurable sweep over the space of supported floating-point formats. The workflow evaluates both fixed-precision and mixed-precision scenarios. Before kernel execution, the script can apply various scaling strategies, including normalization (map data to $[0, 1]$ or $[-1, 1]$), standardization (scaling input data to a distribution with mean = 0 and standard deviation = 1),

Table 5.3: fixed and mixed-precision results relative to the FP32 reference. In the mixed-precision variants, the first two entries denote the data type used for the input data, and the last entry denotes the output type. K-Means kernel only has one input in the mixed-precision: The first entry is the input data type, and the third entry is the output data type.

Kernel	Input Data Features	FP16		FP8			FP8, FP8, FP16			FP8, FP8, FP32			FP16, FP16, FP8			FP8, FP16, FP16			FP16, FP16, FP32	
		MSE	RAE	FP8	MSE	RAE	FP8	MSE	RAE	FP8	MSE	RAE	FP8	MSE	RAE	FP8	MSE	RAE	MSE	RAE
MATMUL	std = 0.01	1.53e-13	7.44e-4	E5M2	9.26e-9	1.72e-1	E5M2	1.85e-9	7.94e-2	E5M2	1.32e-9	7.50e-2	E5M2	1.05e-8	1.85e-1	E5M2	7.01e-10	5.16e-2	**2.26e-14**	**3.09e-4**
	std = 0.5	1.05e-6	7.37e-4	E3M4	3.67e-3	4.65e-2	E3M4	6.17e-4	1.99e-2	E3M4	5.11e-4	1.99e-2	E3M4	3.62e-3	4.52e-2	E3M4	3.07e-4	1.36e-2	**1.38e-7**	**2.86e-4**
	std = 1	1.18e-5	7.13e-4	E3M4	2.32e-1	5.36e-2	E3M4	9.05e-3	1.81e-2	E3M4	8.15e-3	1.72e-2	E3M4	2.04e-1	5.15e-2	E3M4	4.41e-3	1.32e-2	**2.17e-6**	**2.89e-4**
CONV	std = 0.01	2.20e-13	6.84e-4	E5M2	1.22e-8	1.73e-1	E5M2	1.28e-9	9.17e-2	E5M2	1.65e-9	6.98e-2	E5M2	7.34e-9	1.71e-1	E5M2	7.81e-10	5.08e-2	**2.20e-14**	**3.59e-4**
	std = 0.5	5.93e-7	6.89e-4	E3M4	2.78e-3	4.94e-2	E3M4	1.05e-3	2.10e-2	E3M4	5.85e-4	1.90e-2	E2M5	3.42e-3	4.31e-2	E3M4	2.13e-4	1.38e-2	**1.36e-7**	**3.00e-4**
	std = 1	1.65e-5	8.05e-4	E3M4	1.05e-1	4.68e-2	E2M5	3.94e-3	1.36e-2	E2M5	3.80e-3	1.37e-2	E3M4	1.30e-1	5.02e-2	E2M5	2.06e-3	9.45e-3	**2.73e-6**	**3.15e-4**
FIR	std = 0.01	2.28e-13	8.11e-4	E5M2	1.50e-8	2.05e-1	E5M2	1.70e-9	7.25e-2	E5M2	2.47e-9	7.14e-2	E5M2	8.91e-9	1.80e-1	E5M2	1.06e-9	5.09e-2	**3.32e-14**	**2.96e-4**
	std = 0.5	1.04e-6	7.11e-4	E3M4	6.29e-3	5.71e-2	E3M4	7.97e-4	1.96e-2	E3M4	6.72e-4	1.89e-2	E3M4	5.55e-3	5.15e-2	E3M4	2.46e-4	1.43e-2	**1.69e-7**	**2.60e-4**
	std = 1	2.98e-5	7.40e-4	E4M3	4.86e-1	1.13e-1	E2M5	6.93e-3	1.44e-2	E2M5	5.95e-3	1.34e-2	E4M3	3.64e-1	1.03e-1	E3M4	4.67e-3	1.31e-2	**2.33e-6**	**2.75e-4**
K-Means	Normalized scaling	1.06e-7	4.90e-4	E4M3	4.18e-2	2.22e-1	E4M3	4.33e-4	4.56e-2	E4M3	4.33e-4	4.56e-2	E4M3	6.52e-2	6.86e-1	E4M3	4.33e-4	4.56e-2	**1.25e-9**	**9.69e-5**
DWT	Constant scaling	8.33e-9	1.35e-3	E4M3	6.82e-5	1.45e-1	E3M4	4.09e-5	1.39e-1	E3M4	4.09e-5	1.39e-1	E4M3	6.84e-5	1.45e-1	E4M3	1.56e-5	7.41e-2	**1.21e-9**	**6.42e-4**
FFT	Constant scaling	9.81e-10	3.20e-3	E5M2	3.14e-5	6.08e-1	E5M2	8.63e-6	3.11e-1	E5M2	8.63e-6	3.10e-1	E5M2	3.65e-5	6.28e-1	E5M2	2.03e-6	1.37e-1	**3.15e-10**	**2.02e-3**
SVM	RBF kernel	Accuracy (%)			Accuracy (%)			Accuracy (%)			Accuracy (%)			Accuracy (%)			Accuracy (%)		Accuracy (%)	
		94.16		E3M4	80		E3M4	95		E3M4	95		E3M4	84.16		E4M3	95		94.16	

and multiplicative (multiplying input data by a constant < 1). For FP8 formats, the framework also performs an exhaustive search over mantissa bit widths, enabling developers to identify the most suitable exponent–mantissa configuration for a given workload. Each configuration is processed independently, and all error metrics are collected using consistent evaluation criteria. Finally, all results are stored in structured spreadsheet files to support post-processing, visualization, and regression testing.

This automated exploration framework provides a methodology for analyzing transprecision behavior across kernels, enabling designers to compare precision-accuracy trade-offs under different application constraints. By providing this feature, TransLib elevates precision tuning from ad-hoc experimentation to a principled, automated, and hardware-aware exploration process. This design fulfills the need for systematic tools that connect algorithmic precision choices with the constraints of modern parallel IoT end nodes.

5.2 Hardware platforms

We use the PULP-Open (an instance of the PULP architecture) platform and GAP9 as the case studies. To enhance platform compatibility, the framework introduces a target configuration flag within the build system to distinguish between different toolchain environments. By default, the target is set to *pulp*, aligning with the PULP-Open toolchain. To enable execution on GAP9, developers can explicitly specify this target. Moreover, to accommodate the evolving build infrastructure of the GAP SDK, the framework includes dedicated CMakeLists files for each kernel. The inclusion of CMake support facilitates reproducibility and portability across development environments.

5.3 Experimental results

5.3.1 Data Type Exploration

As introduced in Section 5.1.8, the exploration framework systematically evaluates each kernel across standard floating-point formats, including FP8 with varying mantissa widths, using error metrics relative to an FP32 golden reference. In this section, we report the results of this exploration, which traverse the data-type space according to user-specified input sizes, statistics, and scaling methods. The exploration space for each kernel is quite large. For a kernel instance (e.g., MATMUL with given input mean and standard deviation), the number of data-type configurations is:

$$D = |E_8| + 3 \cdot |T|^2 \cdot |E_8| + 3 \cdot |E_8| \cdot |T| + |T|^3 \tag{5.2}$$

where $|E_8| = 6$ is the set of FP8 variants (mantissa bits 1–6) and $|T| = 3$ is the set of non-FP8 types {FP16, FP16alt, FP32}. Substituting these values yields

$$D = 6 + 3 \cdot 3^2 \cdot 6 + 3 \cdot 6 \cdot 3 + 3^3 = 249.$$

Given the size of the design space, we limit the analysis to the most informative configurations. Table 5.3 summarizes fixed-precision results—FP16 and the best-performing FP8 configuration— and five mixed-precision configurations, showing MSE and RAE for all kernels and classification accuracy for SVM. FP16 and FP16alt behave nearly identically in our experiments; to avoid redundancy, we present FP16 only. FP8 formats are denoted $\mathrm{E}x\mathrm{M}y$, indicating x exponent and y mantissa bits [79].

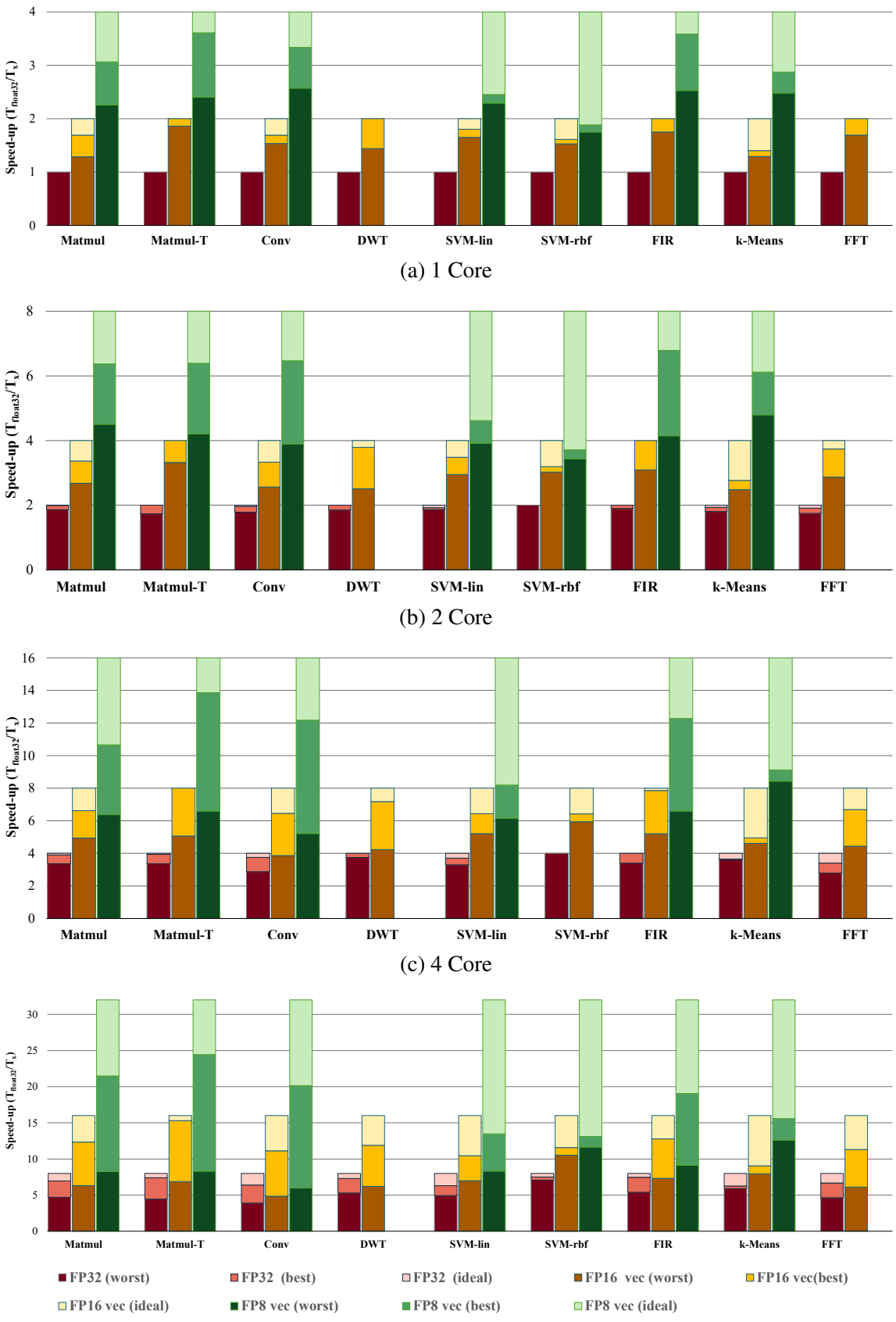
For MATMUL, CONV, and FIR we sweep three zero-mean random Gaussian inputs $\mathbf{x} \sim \mathcal{N}(0, \sigma^2)$ with standard deviation $\sigma \in \{0.01, 0.5, 1\}$. As the standard deviation increases, both MSE and RAE grow, yet FP16 remains well bounded in all cases ($\mathrm{MSE} \leq 3 \times 10^{-5}$, $\mathrm{RAE} \leq 8.1 \times 10^{-4}$). In higher-variance cases, FP8 variants with more mantissa and fewer exponent bits (i.e., higher precision, narrower dynamic range) outperform $\mathrm{E5M2}$—e.g., $\mathrm{E3M4}$ or $\mathrm{E4M3}$ yield lower errors. For K-MEANS, DWT, FFT, and SVM we use the available datasets in TransLib[3]. The inputs are normalized for K-MEANS and scaled by a constant factor ($<$ 1) for DWT and FFT to maintain consistency with the input data range used in other results. Under FP16, errors remain small across these kernels, MSE on the order of 10^{-7} and RAE around 10^{-3}. With FP8, errors increase: MSE rises into the 10^{-5}–10^{-2} range and RAE into the 10^{-1}–6×10^{-1} range, as reported in Table 5.3. For SVM (RBF), FP16 yields 94.16% accuracy, while FP8 reduces it to 80%. Mixed-precision configurations recover the accuracy to approximately 95% in all cases except `FP16, FP16` $\rightarrow$ `FP8`, where it drops to 84.16% due to the loss incurred by downcasting FP16 to FP8, which has less precision and dynamic range.

Across mixed-precision configurations, `FP16, FP16` $\rightarrow$ `FP32` produces the lowest errors, often surpassing FP16. More broadly, mixed precision outperforms FP8 in accuracy, offering a better accuracy–efficiency trade-off. These results provide practical guidance for selecting data formats that meet target accuracy and error metrics.

5.3.2 Fixed-precision on GAP9

We utilize internal performance counters to measure FP32, FP16, and bfloat16 cycles on GAP9. For power consumption measurements, we used Nordic Semiconductor's Power Profiler Kit 2 (PPK2) with a sampling frequency of 100 kHz. The native support for the FP8

[3] *https://github.com/ahmad-mirsalari/TransLib*



(a) 1 Core

(b) 2 Core

(c) 4 Core

(d) 8 Cores

Figure 5.3: Speedup of parallel (1, 2, 4, and 8 cores) and vectorized (float16/bfloat16, float8) kernel versions w.r.t. the float32 baseline (1 core).

data type is unavailable on GAP9, but it is being considered for future iterations of the SoC, and it is already available on its FPU hardware block[4]. Therefore, we measured the latency by adding FP8 support to the event-driven simulator GVSoC[5], which provides cycle-accurate simulations. We therefore approximate the assessment of FP8 energy consumption by executing `FP16` kernels that replicate the same instruction mix and memory traffic as the FP8 variants (i.e., identical counts of loads/stores and multiply–accumulate operations, with matching vector widths and loop bounds). This isolates the effect of data type while preserving datapath and memory-access behavior, yielding a conservative upper bound on the evaluated microarchitecture.

Figures 5.3 and 5.4 illustrate the speedups and energy savings for 1 to 8-core runs, combining the benefits of parallelism and vectorization. The suffix vec indicates the execution of the vector variant. We used the FP32 single-core without vectorization as the baseline for normalization. The bars show the worst, best, and ideal values for all configurations.

5.3.2.1 Parallel speedup and vectorization

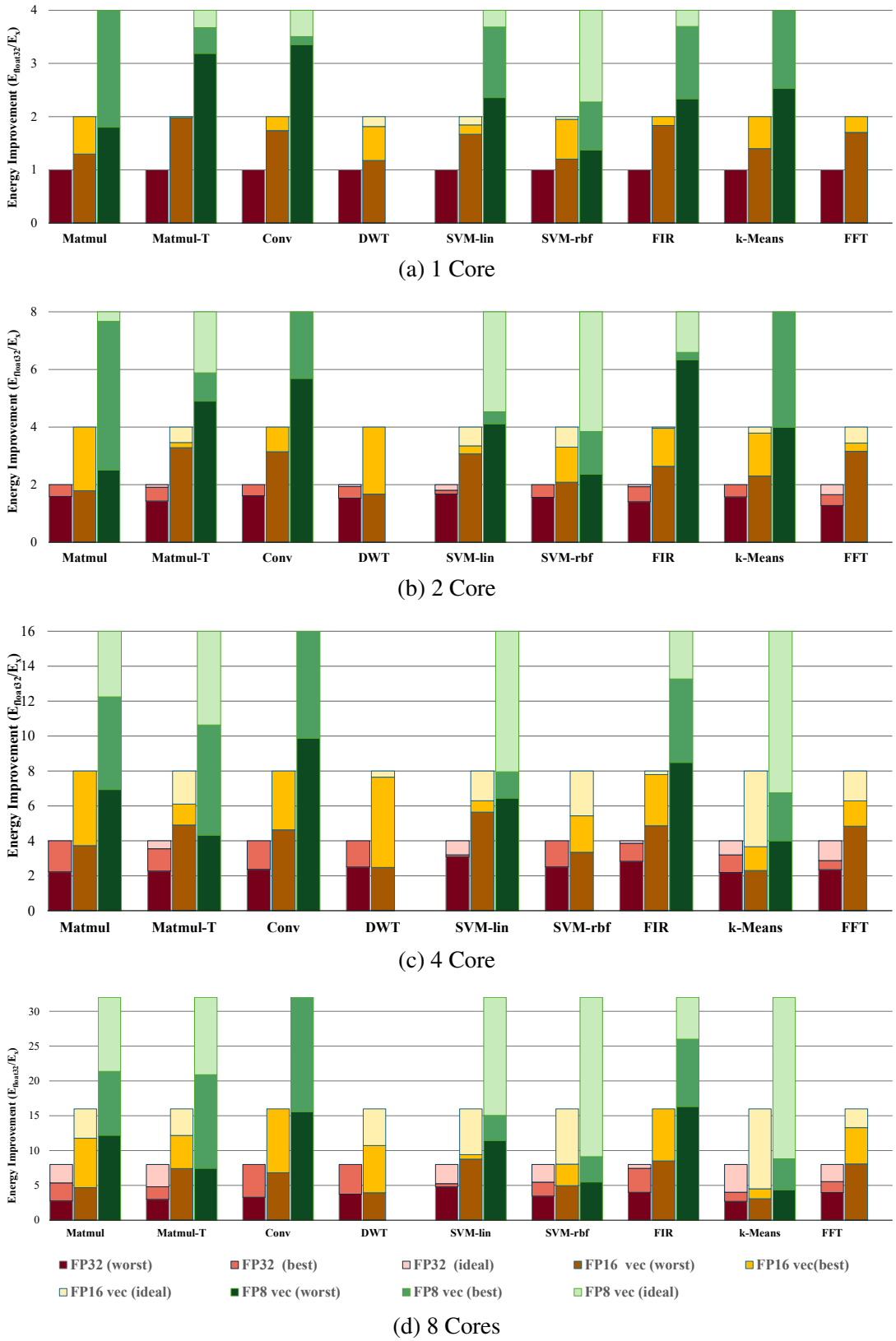
In the absence of vectorization (Fig. 5.3), the parallel speedup reported for all kernels is nearly linear yet sub-ideal because of synchronization, cache contention, and non-parallelizable code regions that limit strong scaling at higher core counts. On average, we achieved $1.97\times$ and, $3.8\times$, $6.9\times$ speedups on 2, 4, and 8 cores, respectively.

Under 16- and 8-bit SIMD vectorization, the theoretical per-lane speedups compared to FP32 are $2\times$ and $4\times$, respectively. In practice, however, the achieved gains are kernel-dependent because vectorization incurs nontrivial overheads, including data packing and alignment, handling of tail (leftover) elements, horizontal (final) vector reductions/accumulations, type conversions, and code regions that remain scalar or inherently sequential. Using 16-bit FP, the smallest speedup appears in K-MEANS, which reaches $1.4\times$ on 1 core and $9.06\times$ on 8 cores, primarily because reductions and other control-intensive sections resist effective vectorization. For FP8, the smallest gain is observed for SVM-RBF at $1.88\times$ on 1 core and $13.2\times$ on 8 cores. The kernel comprises multiple stages—lane-wise vector reductions, control-flow decisions conditioned on these aggregates, and sections that are inherently non-vectorizable. At the other extreme, pre-transposed MATMUL achieves $15.3\times$ with 16-bit FP and up to $24.4\times$ with FP8, illustrating how a regular kernel can fully exploit wide-vector, low-precision execution. By enabling 16-bit vectorization, we can achieve an average of $1.8\times$, $3.5\times$, $6.7\times$, and $11.68\times$ speedups on 1, 2, 4, and 8 cores, respectively.

Reducing operand width to 8 bits increases computational intensity (operations per byte) and lowers memory traffic per operand. However, the realized speedups are highly sensitive to the costs of format conversion to/from higher-precision accumulators, saturation/clamping to manage limited dynamic range, and tail processing for iteration counts that are not multiples of the vector width. On average, enabling 8-bit vectorization can achieve $2.95\times$, $5.68\times$, $10.5\times$, and $17.6\times$ speedups on 1, 2, 4, and 8 cores, respectively

As illustrative examples, we briefly highlight two of the code-level optimizations described in Section 5.1.5. The vectorized MATMUL reaches $12.3\times$, whereas the pre-transposed variant improves to $15.3\times$. For FIR, we adopted a reversed-filter traversal that aligns coefficient access with the input stream, mitigating misaligned loads and branchy edge handling while enabling longer contiguous vector strides. This raises the 16-bit 8-core speedup from $10.8\times$ to $12.7\times$. In some kernels—exemplified by CONV—the speedup is bounded by a serial re-

[4] https://github.com/openhwgroup/cvfpu
[5] https://github.com/gvsoc/gvsoc



(a) 1 Core

(b) 2 Core

(c) 4 Core

(d) 8 Cores

Figure 5.4: Energy saving of parallel (1, 2, 4, and 8 cores) and vectorized (float16/bfloat16, float8) kernel versions w.r.t. the float32 baseline (1 core).

Table 5.4: Execution time of fused mixed-precision (-mfaux) and vectorization (normalized to FP32). In the mixed-precision variants, the first two entries specify the input data type, and the last entry specifies the output type.

Kernel	Fixed-precision		Mixed-precision; Scalar code with no extensions				
	FP16	FP8	FP8, FP8, FP32	FP8, FP8, FP16	FP16, FP16, FP32	FP16, FP16, FP8	FP8, FP16, FP16
MATMUL	1	1	0.71	0.71	0.71	0.71	0.88
FIR	1	1	0.73	0.73	0.73	0.73	0.89
RBF	1	1	0.78	0.79	0.78	0.79	0.79
Scalar version; Using smallFloats mfaux extension for mixed-precision variants							
MATMUL	1	1	1	0.99	1	0.99	Not supported
FIR	1	1	1	0.99	1	0.99	Not supported
RBF	1	1	0.79	0.78	0.81	0.80	Not supported
Vectorization; Mixed-precision vectorization is performed with dot-product-and-sum operations							
MATMUL	2	3.31	2.60	2.56	2.00	2.00	Not supported
FIR	2	3.49	2.43	2.39	1.61	1.61	Not supported
RBF	1.26	1.32	1.35	1.34	1.20	1.18	Not supported

duction that aggregates the element-wise products; this non-parallelizable phase increasingly dominates at higher core counts and thus limits strong scaling. In other cases—such as DWT, FIR, and SVM—multiple synchronization barriers and intrinsically sequential regions required for correctness further limit parallel efficiency.

5.3.2.2 Energy

As shown in Fig. 5.4, the same energy-efficiency patterns emerge. Two effects stand out. First, at a fixed core count, moving from 32 → 16 → 8-bit consistently improves energy efficiency because lower precision increases compute density and reduces memory traffic. Using the FP32 single-core configuration as the baseline, the average energy-reduction factors are 1.95× (16-bit-vec) and 3.47× (8-bit-vec) at one core. Second, multicore parallelism composes with SIMD: as core count increases, time-to-solution falls faster than power grows, so total energy drops further—most notably at eight cores, where the averages reach 6.07× (FP32), 11.27× (16-bit-vec), and 18.65× (8-bit-vec). For FP8 specifically, the realized savings remain slightly below the theoretical maximum because packing/unpacking and conversion overheads are more pronounced: on typical RISC designs, widening FP8 vectors for accumulation often requires multiple instructions where 16-bit unpacking can be done with one.

5.3.3 Mixed-precision on PULP

Table 5.4 quantifies the performance impact of mixed precision in three regimes. When mixed precision is software emulated, the critical path includes explicit operand widening before every multiply–accumulate, so throughput falls below that of fixed-precision. The slowdown is most pronounced when both inputs must be promoted at each MAC —e.g., `FP8,FP8→FP32`— where normalized speed ranges around 0.71–0.79×; when only one operand requires promotion—e.g., `FP8,FP16→FP16` or `FP16,FP32→FP32` —the overhead is smaller and the runtime approaches fixed-precision (0.79–0.89×).

Enabling the -mfaux flag replaces these cast sequences with a single fused mixed-precision accumulation instruction that performs operand promotion and accumulation using fused operations; as a result, mixed precision matches fixed-precision throughput (≈1.00×). Because

Table 5.5: Percentage of memory saving compared to float32. In the mixed-precision variants, the first two entries denote the data type used for the input data, and the last entry denotes the output type.

	Small					Large				
Kernel	FP16	FP8	FP8,FP8,FP16	FP8,FP16,FP16	FP16,FP8,FP16	FP16	FP8	FP8,FP8,FP16	FP8,FP16,FP16	FP16,FP8,FP16
MATMUL	31.82	47.73	42.71	36.18	38.35	45.61	68.41	60.65	53.37	52.89
CONV	14.04	21.07	18.68	18.29	14.44	43.83	64.75	55.53	55.46	43.90
FIR	24.25	36.38	30.69	30.31	24.63	32.66	48.99	41.08	40.82	32.91
	FP16	**FP8**	**FP8,FP16**	**FP16,FP8**	**FP8,FP32**	**FP16**	**FP8**	**FP8,FP16**	**FP16,FP8**	**FP8,FP32**
KMEANS	40.89	61.37	59.26	43.00	55.13	44.98	67.50	65.20	47.20	60.65

the instruction accumulates in FP32, workloads that report a narrower output still perform a final down-cast, which explains the small deviation from unity (0.99–1.00×) when the output is not FP32.

Why `-mfaux` does not yield 1× for RBF? In the RBF kernel, the core arithmetic is structured as difference-then-square:

$$d \leftarrow (T_{\text{out}})a - (T_{\text{out}})b, \qquad d, \texttt{sum} \in T_{\text{out}},\ a, b \in T_{\text{in}}. \tag{5.3}$$

$$\texttt{sum} \mathrel{+}= (T_{\text{out}})d \cdot (T_{\text{out}})d. \tag{5.4}$$

Because both operands are explicitly cast to T_{out} before the multiply, the squaring step is a fixed-precision FMA in T_{out}; `-mfaux`, which fuses mixed-precision MACs only when low-precision multiplicands meet at the multiply and the accumulation targets a different precision, has nothing to compress here. Moreover, when $T_{\text{in}} \neq T_{\text{out}}$, the required casts at the subtraction sites (two per element) still incur overhead, so the kernel does not recover to exactly 1× even though the multiply–accumulate itself is already a single FMA.

Vectorization compounds these effects. For fixed-precision FP8 vectorization, we observe 3.3–3.5× speedup for MATMUL/FIR (≈1.3× for RBF), while mixed-precision vectorization with FP8 operands delivers 2.4–2.6× thanks to dot-product-and-sum operations. With FP16, fixed precision reaches ≈2× (≈1.26× for RBF), `FP16,FP16→FP32` preserves ≈2×, and `FP16,FP16→FP8` attains ≈1.6× due to the terminal down-cast. These results demonstrated that software casts are the primary bottleneck, whereas hardware promotion via -mfaux restores fixed-precision efficiency, and vectorization provides substantial gains.

5.3.4 Memory footprint

Table 5.5 reports the percentage memory savings relative to FP32. The footprint we measure is the sum of the ELF `.data` and .bss segments: .data contains initialized payloads (e.g., inputs, filters/twiddles, constants), while `.bss` accounts for zero-initialized workspaces and output buffers allocated by the kernel/runtime. We evaluate two problem scales—Small and Large—by varying input and filter sizes.

In theory, narrowing payloads saves ≈50% for FP16/BF16 (2× smaller words) and ≈75% for FP8 (4× smaller). In practice, our reductions are smaller because a non-trivial share of the footprint is datatype-invariant (loop/state variables, indices, control structures, and other fixed buffers that live in .bss).

Table 5.6: Fixed- and mixed-precision configurations for the PT ECG pipeline. Tuples indicate formats for (pre-proc, filtering, QRS, HR).

	Fixed-precision			Mixed-precision (vectorized)			Mazzoni [73]
	FP32	**FP16-vec**	**FP8-vec**	**FP16, FP8, FP16, FP16**	**FP8, FP8, FP8, FP16**	**FP8, FP8, FP16, FP16**	**FP32**
Accuracy%	99.92	99.92	99.84	99.85	99.84	99.86	99.53
Max diff (bpm)	0	1.92	32.17	10.13	12.47	8.29	N/A
Average diff (bpm)	0	0.28	7.26	1.33	1.62	1.43	N/A
Speedup	1	1.34	2.06	1.62	2.06	2.16	0.70
Memory saving	1	24.95	37.56	35.11	37.56	37.56	1
Cycles for each sample	1946	1443	940	1200	940	899	2771
Throughput [samples/ms]	87.35	117.81	180.85	141.66	180.85	189.09	61.35

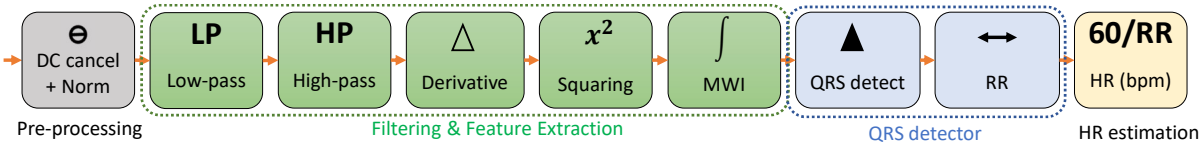


Figure 5.5: ECG processing pipeline: signal conditioning (DC cancel & normalization), filtering & feature extraction (LP, HP, derivative, squaring, MWI), and beat detection & rate estimation (QRS detect → RR → 60/RR(s)).

Moving from Small to Large problems increases savings for every kernel and scheme. For example, MATMUL with FP8 rises from 47.73% (Small) to 68.41% (Large), and CONV with FP8 from 21.07% to 64.75%. This reflects the growing share of activations/workspaces at larger sizes, which benefit more from lower-precision storage.

FP8 achieves the top savings in each setting. Layouts such as `FP8,FP8→FP16` typically trail pure FP8 by a few percentage points (e.g., MATMUL Large: 60.65% vs 68.41%), indicating that reserving FP16 for accumulator buffers has a small impact. Pure FP16 yields 14–46% savings on Small problems and 32–46% on Large ones.

5.3.5 End-to-End Application: Heart Rate Detection

To demonstrate the practical applicability of our library and optimization strategies, we integrate a real-time heart rate detection system based on the Pan and Tompkins (PT) algorithm, originally presented in [73]. This algorithm is widely used in wearable and embedded health monitoring systems due to its balance between computational efficiency and accuracy in detecting R-peaks in electrocardiogram (ECG) signals. As shown in Fig. 5.5, the PT pipeline consists of a sequence of digital signal processing stages: a low-pass filter to suppress high-frequency noise, a high-pass filter to remove baseline drift, a derivative filter to emphasize the QRS complex, a squaring operation to amplify peaks, and a moving window integrator (MWI) to smooth the signal. An adaptive thresholding and decision logic then identifies R-peak candidates. These stages are applied sequentially and rely heavily on convolution-style filtering operations, making the kernel both computationally and memory-intensive, particularly under real-time constraints.

To evaluate this biomedical algorithm under realistic execution models and assess the trade-offs of quantization, we restructured the PT pipeline within our TransLib framework as a fully streaming, modular system. Each stage was implemented using circular buffers and streaming logic to closely mimic the constraints of real-time embedded deployment, including low-latency, low-memory behavior. Precision assignment is configurable per stage/filter, enabling

simulation of both fixed- and mixed-precision pipelines. Each operation in the pipeline—including filter inputs, coefficients, and accumulators—can be independently mapped to different floating-point formats. To enable deeper exploration of FP8 quantization, each stage can use a different format. The pipeline exports filtered signals and detected R-peaks to C header files, enabling direct integration with different implementations.

Following Mazzoni et al. [73], detection accuracy is evaluated on the R–R (RR) interval sequence, not on heart-rate (HR) values. Specifically, the RR intervals produced by the detector are compared with a golden reference obtained using the MATLAB *findpeaks* function, and accuracy is computed from the RMSD of RR intervals and their normalized complement. Thus, correctness is judged by how well R-peak timing (hence RR) is recovered, rather than the downstream HR estimate.

To make the impact of numeric quantization visible at the application level, in addition to the RR-based metric above, we also report the maximum absolute HR deviation between the quantized and floating-point pipelines:

$$\Delta\mathrm{HR}_{\max} = \max_{t\in\mathcal{T}} \left|\mathrm{HR}_{\mathrm{quant}}(t) - \mathrm{HR}_{\mathrm{float}}(t)\right|. \tag{5.5}$$

Consistent with [73], we partition the processing into four blocks for controlled quantization sweeps as shown in Fig. 5.5: (i) signal conditioning (DC cancellation and normalization); (ii) filtering & feature extraction (low-pass, high-pass, derivative, squaring, and moving-window integration); (iii) QRS detector (adaptive dual-threshold with search-back); and (iv) HR estimation. To simplify the exploration, we report one quantization configuration for the Filtering & Feature Extraction block. Table 5.6 summarizes the high-level algorithmic metrics and hardware results, including RR-based accuracy, HR deviation, speedup, throughput, and memory usage. All hardware measurements were obtained on a PULP-Open implementation clocked at $170\,\mathrm{MHz}$ to compare throughput with our baseline [73]. All values refer to FP32 as the baseline.

Among fixed-precision baselines, FP16-vec attains 99.92% accuracy with low HR error (max 1.92 bpm, avg 0.28 bpm), a 1.34× speedup and 117.81 samples/ms throughput, while saving ≈25 % memory. FP8-vec preserves high detection accuracy (99.84%) but increases HR deviation (max 32.17 bpm, avg 7.26 bpm); in return, it delivers a 2.06× speedup, 180.85 samples/ms throughput, and ≈37.56% memory saving. The Pan–Tompkins chain detects beats primarily from the shape and timing of the integrated envelope with adaptive thresholds, which makes it largely invariant to amplitude quantization; small FP8 amplitude errors rarely change the peak index, so RR intervals—and thus the RR-based accuracy metric—remain high. However, HR is a nonlinear transform of RR, $\mathrm{HR} = 60/\mathrm{RR}$. A tiny timing error $\Delta\mathrm{RR}$ is amplified in bpm according to:

$$\frac{\mathrm{d}\,\mathrm{HR}}{\mathrm{d}\,\mathrm{RR}} = -\frac{60}{\mathrm{RR}^2} \Rightarrow \left|\Delta\mathrm{HR}\right| \approx \frac{60}{\mathrm{RR}^2}\left|\Delta\mathrm{RR}\right|. \tag{5.6}$$

Consequently, the same $\Delta\mathrm{RR}$ produces larger HR deviations at higher heart rates (smaller RR). The reported memory savings (including the ELF .data and .bss segments) fall short of the ideal scaling: vectorization requires additional buffers, and parts of the QRS logic maintain INT32 state. As a result, moving from FP32 to FP16/FP8 does not halve memory usage, even though substantial reductions are still achieved.

Mixed-precision configurations temper FP8's HR deviation without sacrificing performance. Setting HR estimation in FP16 while keeping the preceding blocks in FP8 (FP8, FP8, FP8, FP16) yields 99.84% accuracy with markedly lower HR error (max 12.47 bpm, avg 1.62 bpm), 2.06× speedup, and 180.85 samples/ms throughput, with ≈37.56% memory saving.

Promoting QRS and HR estimation to FP16 (FP8, FP8, FP16, FP16) achieves the best overall trade-off: accuracy 99.86%, the lowest HR error among mixed setups (max 8.29 bpm, avg 1.43 bpm), the highest speedup (2.16×), and the highest throughput (189.09 samples/ms), with the same ≈37.56% memory reduction. The performance edge over FP8 likely stems from fewer and cheaper `FP8`↔`INT32` conversions within the QRS stage and reduced timing when computing RR intervals.

5.4 Conclusion and Future Work

This chapter introduced TransLib, an open-source library and workflow for studying reduced- and mixed-precision floating-point kernels on MCU-class systems. TransLib couples Python reference models with optimized C implementations and exposes principled controls for storage and compute precision, vector width, and parallel execution. To make mixed precision practical in tight inner loops, we proposed two compiler intrinsics—a load–convert for packed widening/narrowing under a defined rounding mode, and a fused mixed-precision multiply–accumulate that decouples memory formats from arithmetic while avoiding conversion overheads.

Our experimental evaluation on PULP-Open and GAP9 shows that precision and parallelism can be traded for efficiency without sacrificing task-level quality. In particular, 16-bit SIMD delivers substantial gains while preserving accuracy near round-off, and FP8 SIMD further increases throughput with sizable memory savings. Pure software emulation of mixed precision reduces throughput, but the adoption of fused operations recovers fixed-precision performance for multiply-and-accumulate patterns. Overall, reduced precision yields notable reductions in memory footprint and, when supported end-to-end by the hardware, brings consistent speedups from both SIMD and multicore scaling. We instantiated TransLib kernels in a real ECG detection pipeline to demonstrate applicability.

TransLib thus provides a hardware-aware methodology for selecting numeric formats that meet end-to-end accuracy while improving execution time and memory usage on resource-constrained platforms. Future work includes automated precision assignment (per-kernel and per-operator), broader integer/fixed-point back ends (including block-floating and logarithmic formats), native support for emerging FP8 hardware and vector extensions, and expansion to additional sensing and signal-processing applications.

CHAPTER

6

SOM: A DOMAIN-SPECIFIC APPLICATION FOR REDUCED-PRECISION ARITHMETIC

Pathogenic bacteria pose a significant threat to human health. Consequently, there is a growing demand for precise and efficient methods to rapidly identify bacterial species. Portable, low-cost genomics platforms for edge computing can enable timely identification and containment of outbreaks in remote areas, support widespread genome testing and personalized treatment, facilitate frequent monitoring of genomic changes, and improve privacy by avoiding the transfer of raw data to the cloud [30].

However, severe challenges arise when performing genomic computations on embedded systems. Edge Artificial Intelligence (AI) architectures are a promising target, but they are typically limited in terms of computational power and memory capacity. Traditional genomics algorithms are designed for high-performance computers and may exhibit slow and unscalable performance on edge devices [30]. Pathogen identification requires processing a large amount of data, depending on the identification method and the pathogen's complexity.

Artificial Neural Network (ANN) algorithms hold the potential to address the challenges above, offering the prospect of improved speed, accuracy, and reduced bias compared to traditional approaches. Self-Organizing Maps (SOMs) are a class of unsupervised learning models that reduce the dimensionality of data and facilitate clustering of similar data points [51]. In our context, SOMs encapsulate a compressed representation of genomic data that can distinguish between different pathogens without processing the entire sampled DNA. For instance, the algorithm proposed in [136] uses 40k random fragments of the DNA sequence of two strains of E. Coli bacteria to train two SOM networks to classify subsequent sequences of the bacterial strains. One of the core benefits of this approach is its ability to work with DNA fragments rather than requiring fully assembled DNA, as has been attempted by [62] previously.

As a case study of smallFloats, we address the challenges of performing genomics computations for pathogen identification on embedded systems with limited computational power.

6.1 Related Work

Leveraging hardware acceleration is an alternative approach to enhance the efficiency of computationally intensive algorithms. Previous research by [136] has proposed using SOM to accelerate genome identification. The CGRA implementation [116] observed a less than 1% quality loss when training SOM networks using 16-bit fixed-point number representations, compared to the 32-bit FP implementation. The authors of [116] also demonstrated that low-bit fixed-point formats, such as 8-bit, are inadequate for genome identification experiments, as they fail to meet the required accuracy and dynamic range for successful identification. A trend in computing is to customize the precision of floating-point arithmetic in applications to match their specific requirements and constraints within their respective domains [77]. With the increasing computational requirements and dynamic nature of target algorithms, there is a growing need for hardware support for arithmetic operations in smaller-than-32-bit FP formats to enhance system energy efficiency while maintaining the desired level of accuracy [69].

This chapter leverages the features of the GAP9 SoC to propose a portable, low-power solution for genome identification.

6.2 Background

This section provides an overview of the implemented SOM algorithm.

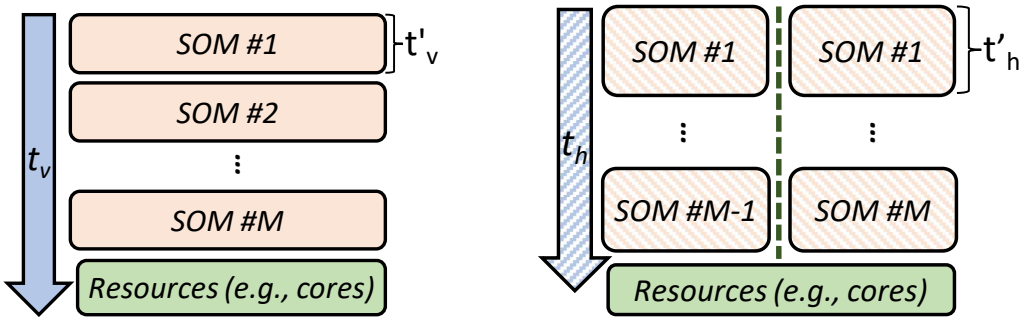
6.2.1 SOM algorithm

Algorithm 1: The computational kernel of SOM training; N: number of neurons, I: input vector, IS: set of input vectors (training samples), W: weight matrix.

for $I_k \in IS$ **do**
  $dist_{min} = \min_{j=1...N}(\sum_{i=1}^{M} |I_{k,i} - W_{i,j}|)$;
  $j_{min} = j$ where $dist_j = dist_{min}$;
  for $j \in 1...N$ **do**
    $dist = ||j - j_{min}| - \frac{N}{2}|$; // toroid distance
    $W_j = W_j - \frac{\beta}{2^{dist}}(W_j - I_k)$;
  end
  $\beta = \max(\beta * decay\_factor, \beta_{min})$; //decay β
end

This chapter uses a circular SOM for bacterial genome recognition described in [136]. Algorithm 1 summarizes the main computational steps. The training set is denoted by $IS = \{I_1, \ldots, I_S\}$, where each $I_k \in \mathbb{R}^M$ is an input vector obtained by segmenting the genome into fixed-length fragments. The goal is to capture the genetic signature of a particular bacterium using a specific SOM; this implies training one SOM per bacterial strain of interest. By evaluating unknown DNA fragments against the trained SOMs, we identify the strain whose SOM yields the closest match. This correlation allows us to identify the unknown bacteria.

A SOM is represented by a weight matrix, denoted as W, with dimensions N × M, where N represents the number of neurons, and M represents the number of weights for each neuron. Each bacterial strain has its dedicated SOM, trained using strain-specific sequences (IS) composed of M-element vectors, with each element representing a nucleotide (C, G, T, or A).



(a) The vertical pattern (b) The horizontal pattern

Figure 6.1: Mapping patterns

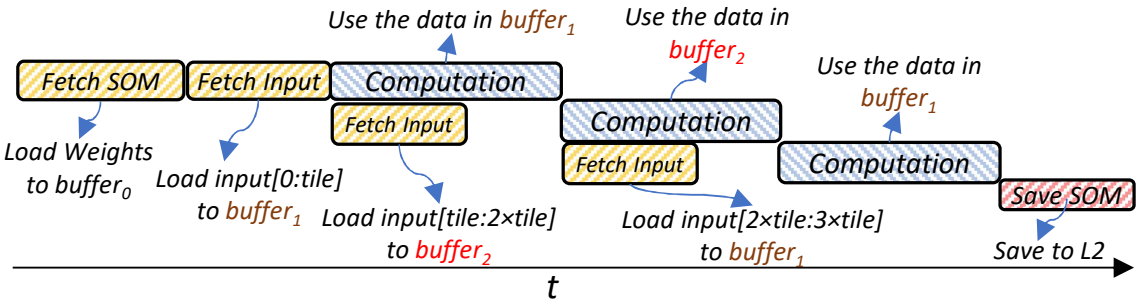


Figure 6.2: Example of tiling and double buffering

These vectors correspond to the short reads obtained from sequencing machines. During training, a random fragment is selected from the sequence fragments corresponding to a bacterial genome. The distance of this fragment from all neurons is computed using the equation stated in line 7. Subsequently, the weights of the winning neuron and its neighbouring neurons are updated based on lines 9-12.

6.3 Methodology

This section presents the details of our methodology.

6.3.1 Computation Mapping

Figure 6.1 depicts two mapping strategies, namely Vertical and Horizontal, used to map the SOM to the GAP9 SoC. In the Vertical pattern, SOM training executes sequentially, with all available resources (i.e., cores and memory) dedicated to a single SOM. This approach is effective when the total workload, determined by factors such as the input size (line 6), the Number of Neurons (line 7), and the neighbourhood radius (line 9), can be efficiently parallelized across all cores. However, if the workload is insufficient, this method does not fully utilize the available cores. Consequently, in the Horizontal pattern, the available resources are divided and shared between the two SOMs being trained. We verified that using more than 2 SOMs with 8 available cores increases overhead without real performance benefits. This approach allows for a more balanced utilization of resources and can potentially improve the overall efficiency of the training process. We will show the results of our experiments in Section 6.4.

To facilitate the execution of large networks that do not fit within the TCDM, we support tiling and double buffering for both SOM parameters and input data. The tiling strategy divides the data into smaller fragments, enabling them to fit within the available amount of TCDM memory. Additionally, this strategy enables seamless data movement between memory levels by utilizing DMA-assisted double buffering. By employing this technique, the data for the next tile can be transferred in parallel with computation on the current tile, reducing the overhead associated with the memory hierarchy. Ideally, this technique allows complete overlap between data transfers and computation phases, as shown in Figure 6.2. To simplify the explanation, assuming the TCDM capacity is sufficient to hold the SOM networks, we demonstrate the use of double buffering to efficiently transfer input data to the TCDM.

6.3.2 FP8

We extended the PyTorch backend to support the FP8|e5m2 data type. The PULP architecture currently supports this format at the hardware design and compiler level. This extension allows us to evaluate the impact of FP8 on memory footprint and performance, and the implementation is available as open source [1].

6.3.3 Vectorization

Despite the inherent limitations of vectorizing the SOM algorithm, we support SIMD vectorization for both proposed computational patterns. In lines 7 and 8, SIMD vectorization is used to calculate the distance between the input and weight values. In line 10, the toroid distance is computed for each neuron relative to the winning neuron. This distance serves as the basis for subsequent operations, including the division operation $\frac{\beta}{2^{dist}}$. In the following step, we utilize a built-in function to generate a vector containing the division results. For each smallFloat format, the compiler provides a specific built-in function to create the corresponding vector with a minimum set of assembly instructions (1 for FP16/bfloat16, 2 for FP8). As a result, vectorization can be applied to all operations on line 11.

6.3.4 Algorithm and Architectural Optimizations

Given that the denominator in the division operation is a power of two, we optimized the computation by converting it into a multiplication operation; $\frac{\beta}{2^{dist}}$ is equal to $\beta \times 2^{-dist}$. We implemented an optimized version of the function that computes the power of two using bit-level manipulation of the exponent. To determine the exponent value for 2^{-dist} in the target format, we subtract the input value (x) from the bias. Then we move this value to the exponent field using a left shift. These steps require only two cycles on the PULP architecture; since there is a single register file for integer and FP data, bit manipulation does not incur register-copy overhead. Using a similar approach, the $fabs$ function has been optimized to efficiently compute the absolute value of a floating point variable x through a $bitwise$ AND operation with a $bitmask$ designed to clear the $sign$ bit. Both optimizations can be generalized to the vector case.

We skip the inner loop as an additional optimisation when the distance exceeds a predefined threshold value. This optimization is possible because the value of 2^{dist} becomes unrepresentable in certain cases. The range limitations of the selected FP data type determine this threshold value. In practical terms, this optimization guarantees that only the winning neuron

[1] https://github.com/ahmad-mirsalari/SOM-on-PULP

and its neighbouring neurons are updated, preceding and following it in a circular SOM within a radius of 2 times the threshold.

6.4 Experimental Results

6.4.1 Experimental Setup

6.4.1.1 Golden Model

We developed a Python reference based on PyTorch to establish a reliable golden model. This model supports various floating-point types, including FP32, FP16, bfloat16, and FP8. We introduced a set of dedicated flags into the golden model to simulate the behaviour of the instructions commonly found in ISA extensions for edge AI nodes. The MAC flag emulates the fused multiply-add (FMA) operator, which is typically available in DSP instruction sets for embedded devices. The vector flag simulates using SIMD vector instructions. Introducing these flags aims to account for the precision changes in intermediate results caused by rounding effects when using the respective instructions (MACs and vector operations) instead of the original ones (mul+add and scalar operations). As a result, the golden model considers the impact of these processor-level operations to ensure accurate and reliable results.

We implemented the SOM algorithm in plain C code, including a set of preprocessor directives to enable vectorization, multi-core processing, and performance monitoring. The full project codebase is available in an open-source repository[2].

6.4.1.2 CGRA fabric

We compare the SOM implementation on GAP9 with a CGRA that targets dense linear algebra applications, including SOM [136]. The authors employed two CGRA fabrics, namely the Dynamically Reconfigurable Resource Array (DRRA) [109] for dense linear algebra and the Distributed Memory Architecture (DiMArch) [124] for streaming scratchpad memory, connected via a configuration network-on-chip (NoC), as described in [136]. Each DRRA cell includes a 16-bit fixed-point arithmetic Data Processing Unit (DPU), a register file, and a sequencer responsible for cell configuration. The arrangement consists of multiple columns, with each column comprising two DRRA cells and two DiMarch cells. The study observed that increasing the number of columns led to a corresponding increase in speedup. The details of SOM mapping, timing, and cell configuration can be found in [136]. As part of their extended work, the authors investigated the impact of different fixed-point bit-widths on the performance of the SOM algorithm [116].

The SOM algorithm was implemented with four different dimensions: 128, 256, 512, and 1024 neurons, and each neuron has 8 weights. This setup allowed for a fair comparison between the PULP implementation and the CGRA. In the evaluated configuration, the CGRA baseline does not exploit features such as SIMD vectorization, DMA-based double buffering, or near-threshold operation, and it does not include the algorithmic optimization described in Sec. 6.3.4.

[2] https://github.com/ahmad-mirsalari/SOM-on-PULP

6.4.2 Accuracy

We trained 10 different SOMs, each with a distinct DNA sequence corresponding to a different bacterium. For each SOM, we used a set of 20,000 training vectors. After the training phase, we utilized the trained networks to identify 1000 unknown DNA sequences. To evaluate network performance, we use the classification error metric [116]. This metric is calculated by dividing the number of times the network misclassified the bacterial strain (C_{false}) by the total number of tests performed (C_{total}):

$$classification\_error = \frac{C_{false}}{C_{total}} \times 100\%$$

Our experimental results demonstrate that the FP16 and bfloat16 formats successfully passed the experiment across different neuron counts, maintaining the same accuracy as the FP32 baseline. However, when using the FP8 format, we observed a classification error of less than 9% with 128 neurons. Notably, this error approaches zero as the number of neurons increases to 256 or higher.

Among fixed-point data formats, the 8-bit format proved ineffective across different networks, yielding unsatisfactory classification results. The 12-bit format exhibits a classification error of 39%. Finally, the 16-bit and 32-bit formats successfully pass the experiment with 0% error.

6.4.3 Hardware Results

In the hardware experiments, we trained two SOM networks with 128, 256, 512, and 1024 neurons. These networks were trained on a dataset of 40,000 training vectors representing two strains of E. coli.

6.4.3.1 Memory footprint and silicon area

Table 6.1 reports the memory footprint required for storing the weights of two SOM networks and the corresponding silicon area requirements. Employing FP8 can substantially reduce memory requirements, up to $2\times$ compared to 16-bit formats, for networks with 256, 512, and 1024 neurons while maintaining accuracy. Additionally, opting for FP8 with 128 neurons allows for further reduction, although at the cost of a 9% decrease in SOM accuracy.

To ensure a fair assessment, comparing the cluster domain of GAP9 (including the TCDM) with the CGRA fabric (excluding the DRAM) is more appropriate. This means we do not consider the L2 memory in GAP9 or the DRAM in the CGRA fabric, as they are responsible for storing the input data during area measurement. In our study, we provided the area measurements for the CGRA fabric with the 8 columns. We can use a CGRA fabric with a 16-bit fixed-point format and 8 columns to ensure a fair comparison in terms of accuracy and maximum speedup. Since the CGRA fabric is implemented in 28 nm, we scaled the area number by 0.6. As a result, GAP9 achieves a $1.13\times$ reduction in the area compared to the CGRA fabric. This increase in area in the CGRA fabric is due to each cell having its own sequencer with dedicated memory.

6.4.3.2 Parallelization and vectorization performance

Figure 6.3 illustrates the speedups achieved executing on 1, 2, 4, and 8 cores, combining the benefits derived from parallelism and vectorization w.r.t the FP32 baseline. The suffixes Vec,

Table 6.1: Memory required for storing the weights of two SOM neurons in bytes and Area in mm^2

	Weight Memory (Byte)				Area (mm^2)	
Neuron / Bit	**128**	**256**	**512**	**1024**	CGRA 8 columns 28nm	GAP9
8-bit	2048	4096	8192	16384		
16-bit	4096	8192	16384	32768	2.79	1.48
32-bit	8192	16384	32768	65536		

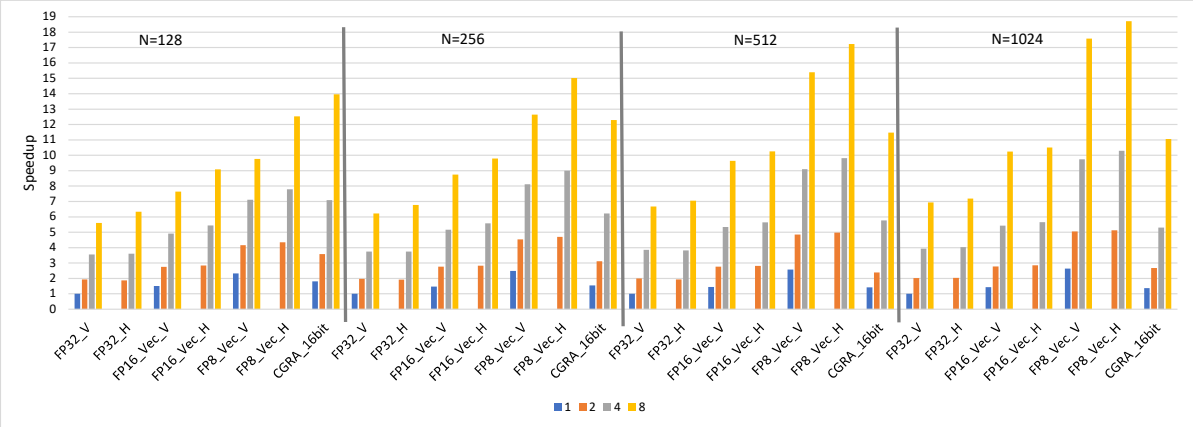


Figure 6.3: Speedup of parallel (2, 4, 8 cores), vectorized (FP16/bfloat16, FP8), and CGRA fabric (1, 2, 4, and 8 columns) compared to the FP32 baseline. V and H represent vertical and horizontal mapping, respectively.

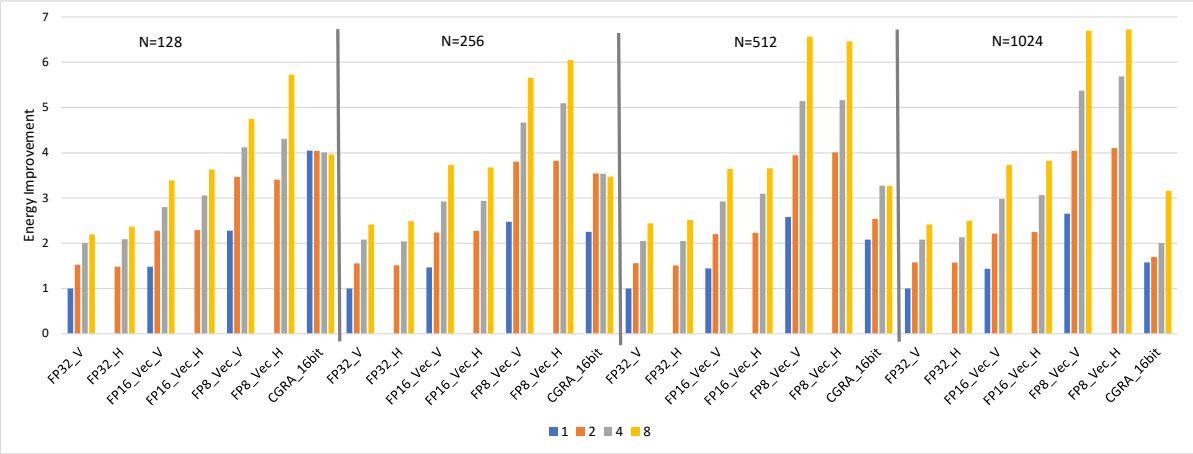


Figure 6.4: Energy improvement of parallel (2, 4, 8 cores), vectorized (FP16/bfloat16, FP8), and CGRA fabric (1, 2, 4, and 8 columns) compared to the FP32 baseline. V and H represent vertical and horizontal mapping, respectively.

V, and H indicate execution using vectorization, Vertical mapping, and Horizontal mapping, respectively. In the same chart, we provide the speedup values for the CGRA fabric with 1, 2, 4, and 8 columns. The baseline cycle counts for the FP32 data type are as follows: 425,180,568 cycles for 128 neurons, 738,577,694 cycles for 256 neurons, 1,364,078,880 cycles for 512 neurons, and 2,613,552,914 cycles for 1024 neurons. On average, we achieved 1.99×, 3.879×, and 7.01× speedups on 2, 4, and 8 cores, respectively, without vectorization using Horizontal mapping.

Notably, the Horizontal mapping approach cannot be used with a single core, as we divide the cores between two SOMs in this configuration. As depicted in Figure 6.3, the Horizontal method consistently achieves higher speedups than the Vertical method, especially when scaling up to 8 cores. As outlined in Section 6.3.4, the determination of a threshold value plays a crucial role in ensuring an accurate representation of 2^{dist}, considering the data range limitations of the chosen FP data type. This threshold value is calculated using the expression $\log_2(max\_number)$, resulting in a maximum threshold value of 19 for FP32 or bfloat16. Notably, despite the different data ranges of FP16/FP8 and FP32/bfloat16, the same threshold can be used across these data types without compromising accuracy. Consequently, the maximum number of neurons to be updated is 39, encompassing the winning neuron and its neighbouring sides in the toroid topology (19 + 1 + 19). In the case of Vertical mapping, these neurons are distributed across 8 cores, resulting in an inadequate workload due to the overhead of parallelization. Conversely, with Horizontal mapping, two SOMs are updated using this number of cores, resulting in a higher speedup.

As depicted in Figure 6.1, the Horizontal mode exhibits a longer runtime for each individual SOM (utilizing half of the available resources) compared to the Vertical mode ($\acute{t}_h > \acute{t}_v$). However, since we simultaneously execute two SOMs in the Horizontal mode, the overall time required is actually shorter than that of the Vertical mode ($t_h < t_v$). On average, vectorization yields a speedup improvement of $1.45\times$ for FP16/bfloat16 and $2.49\times$ for FP8 compared to FP32. As discussed in Section 6.3.3, the SOM algorithm poses challenges in exploiting vectorization.

When comparing the performance of FP16/bfloat16 on GAP9 with the 16-bit fixed-point on CGRA fabric, we observed that in smaller networks, specifically with 128 and 256 neurons, CGRA consistently demonstrated a higher speedup across various column configurations. For example, when using 8 columns, CGRA achieved a speedup of $12.3\times$, while GAP9 achieved a speedup of $9.78\times$ with 8 cores.

However, as the network size increased, GAP9's advantages became more pronounced. For instance, using 1024 neurons, CGRA fabric achieved a speedup of $10.89\times$ with 8 columns, while GAP9 achieved a close speedup of $10.50\times$ with 8 cores. These differences in GAP9 and CGRA stem from the fact that for smaller SOMs, the architectural and algorithmic optimizations do not show up; for larger networks, these advantages play a significant role.

In the case of the FP8 data type, our analysis in Figure 6.3 revealed that CGRA outperformed FP8 in terms of speedup only for 128 neurons and 8 columns, where it achieved a speedup of $13.97\times$ compared to $12\times$ for FP8. However, FP8 consistently achieved higher speedups across other network sizes in all core configurations. For example, with 256, 512, and 1024 neurons, CGRA achieved $12.3\times$, $11.47\times$, and $10.89\times$ speedups with 8 columns, while GAP9 achieved $14.77\times$, $17.19\times$, and $18.60\times$ speedups with 8 cores.

6.4.3.3 Energy

The energy estimation for the CGRA was based on post-layout synthesis simulations in 28 nm technology at 180 MHz. The values are then scaled down to 22 nm for fair comparison using a factor of 0.7 based on [19]. The energy includes DRAM accesses as in [88].

The baseline energy consumption values for the FP32 data type are as follows: 32.17 mJ for 128 neurons, 56.33 mJ for 256 neurons, 104.05 mJ for 512 neurons, and 200.75 mJ for 1024 neurons.

As depicted in Figure 6.4, in the case of a small network, we find that FP16 on GAP9 exhibits a comparable energy improvement when utilizing 8 cores, compared to the 16-bit fixed-point on CGRA fabric. Specifically, FP16 achieves a notable energy improvement of $3.67\times$ using 8 cores, which is very close to the energy improvement achieved by the 16-bit fixed-point at $3.47\times$ in a network with 256 neurons.

However, as network size increases, FP16 shows its advantage, achieving greater energy savings with fewer cores. For instance, with 1024 neurons, FP16 achieves a $3\times$ energy improvement with only 4 cores on the GAP9 platform, surpassing the performance of 16-bit fixed-point, which achieves a $2\times$ improvement on the CGRA fabric.

In the case of FP8, we observe consistently higher energy improvements across all configurations compared to the 16-bit fixed-point implementation on the CGRA fabric. This holds true for networks with 256, 512, and 1024 neurons, and FP8 even surpasses the 16-bit fixed-point approach for 128 neurons when using only 4 cores.

For instance, with a single core, FP8 achieves energy improvements of $2.47\times$, $2.58\times$, and $2.65\times$ in networks with 256, 512, and 1024 neurons, respectively. In contrast, the CGRA fabric yields energy improvements of $2.25\times$, $2.08\times$, and $1.57\times$ when using 1 column in the aforementioned neurons. Considering 8 cores, FP8 demonstrates substantial energy improvements of $6\times$, $6.56\times$, and $6.72\times$ in networks with 256, 512, and 1024 neurons, respectively. Conversely, the CGRA fabric provides energy improvements of $3.47\times$, $3.26\times$, and $3.15\times$ when utilizing 8 columns in the same networks.

6.5 Conclusion and Future Work

This chapter proposes an optimized implementation of SOMs for genomics computations on resource-constrained embedded systems. We explore the use of smallFloat formats to improve energy efficiency and performance. Our approach includes parallelization, data transfer optimization, and comparison with a CGRA, achieving state-of-the-art performance for bacterial identification. We contribute to efficient genomics computations on embedded systems, considering computational limitations and energy constraints. As a significant outcome, we demonstrated that a general-purpose parallel platform can achieve state-of-the-art results for genome identification. Other architectures, such as a CGRA, can achieve better results under the same setup. However, the PULP architecture enables algorithm designers to achieve competitive results across many application contexts by conducting software exploration and leveraging commercially available solutions.

Using a scaling factor is a widely adopted approach to achieve higher-precision values within a range that aligns well with the representable range [78]. As the FP8$|E5M2$ format did not achieve 0% classification error with 128 neurons, we will explore dynamic scaling techniques to enhance FP8's accuracy as future work. Furthermore, we will investigate the implementation of alternative FP8 encodings, such as FP$|E4M3$, to assess their impact on accuracy.

CHAPTER

7

STREAMEASE: STREAM-ORIENTED CONVERSION OF TEMPORAL CONVOLUTIONAL NETWORKS FOR EDGE AI

Streaming applications are designed to process real-time data streams in domains such as autonomous systems [36], video streaming [34], speech processing [65], and biosignals monitoring [117]. Traditional approaches rely on static rule-based systems or signal-processing pipelines processing data sequentially [64]. However, the increasing complexity and variability of real-time data streams have led to the adoption of machine learning (ML) and artificial neural network (ANN) methods. In recent years, Deep Neural Networks (DNNs) have become predominant [81], and designers have proposed several architectures: Recurrent Neural Networks (RNNs) [81], Long Short-Term Memory Networks (LSTMs) [38], and Temporal Convolutional Networks (TCNs) [60].

TCNs leverage dilated convolutions to capture long-term temporal dependencies [10]. Unlike RNNs and LSTMs, TCNs process sequences in parallel, enabling faster training and inference [95]. Moreover, they typically require fewer parameters and provide stable gradients during training [43]. A TCN employs dilated 1-D convolutions to create a temporal receptive field, including the input timesteps contributing to the output at a given point [60]. TCN-based models have demonstrated strong performance across multiple tasks [68] [67].

Advanced normalization techniques [113] [16] further improved the accuracy of TCN-based models. Conv-TasNet [67] employs global layer normalization (gLN) and cumulative layer normalization (cLN) to normalize inputs across the entire sequence. Despite these strengths, TCNs face significant challenges when deployed in real-time environments, especially on resource-constrained devices, due to substantial computational demands [127] [43]. RT-TCN [48] proposes to reduce computational and memory requirements by reusing outputs from previous convolutions. However, this solution has limited applicability and does not

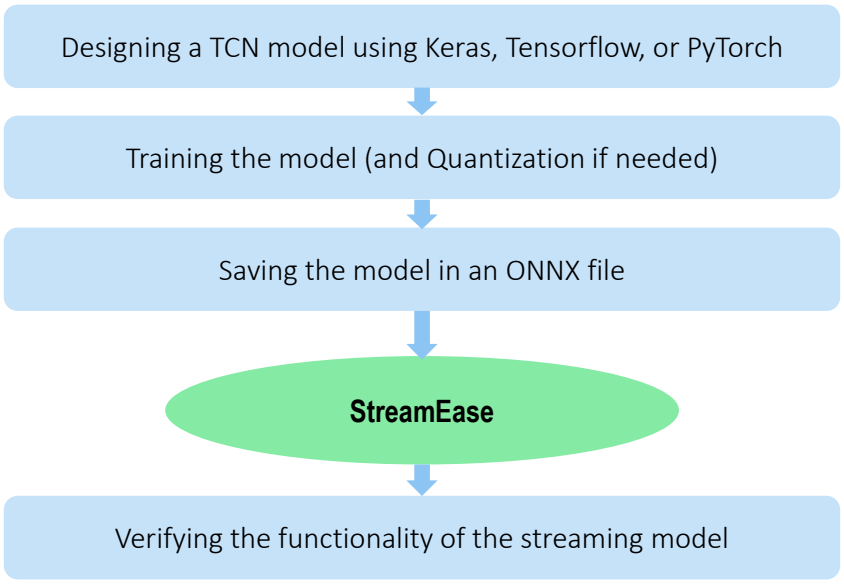


Figure 7.1: End-to-end StreamEase pipeline

support complex network architectures on low-power MCUs.

This chapter proposes a comprehensive optimization strategy for TCN-based models targeting real-time applications on resource-constrained ultra-low-power microcontrollers[1]. As a case study, we provide a detailed evaluation of Conv-TasNet for real-time speech enhancement, deploying the model on GAP9 with 8-bit integer (INT8) and 16-bit floating-point (FP16, bfloat16 (BF16)) Multiply-Accumulate (MAC) vector units. In the rest of this chapter, we define non-streaming models as those that process the entire receptive field each timestep, and streaming models as those that process only the current timestep.

The key contributions of this tool are:

- a dynamic multi-timestep streaming approach for TCN-based real-time applications by processing multiple input timesteps simultaneously. This approach balances computational efficiency and latency.
- a streaming cLN methodology, allowing the model to maintain high performance while operating in streaming mode. This approach enables effective normalization in real-time without relying on future context.
- advanced quantization techniques, including FP16, BF16, and INT8-BF16 mixed precision, reducing the memory footprint while maintaining high performance.

7.1 Methodology

As depicted in Fig. 7.1, we automate the conversion of non-streaming TCN models into streaming ones, allowing developers to construct and train TCN models within any framework, save them in ONNX format, and automatically convert them into streaming ONNX models. We have developed a library that facilitates the automatic conversion of TCN models to achieve this goal. This approach eliminates manual model rewriting, reducing the time and effort

[1]Code available at: https://github.com/ahmad-mirsalari/StreamEase

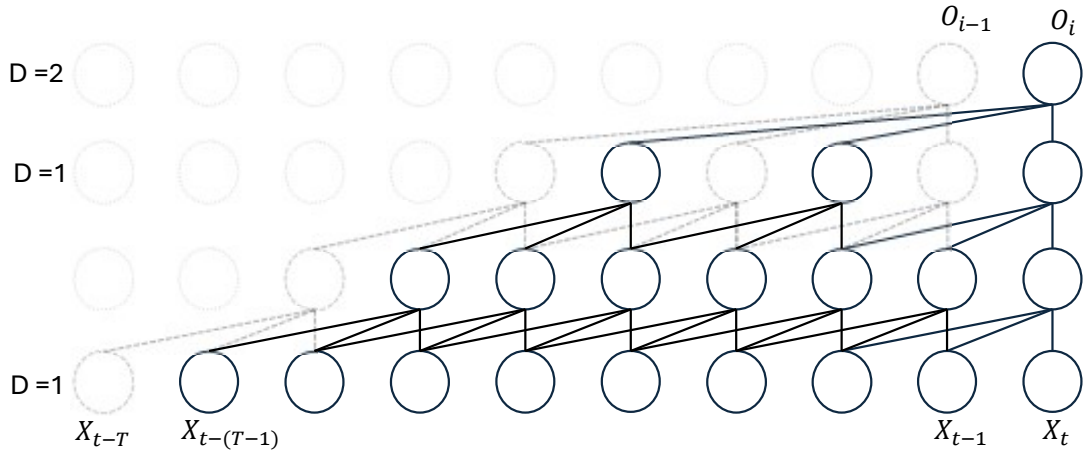


Figure 7.2: Real-time execution of a TCN. The dilation rate of each layer is indicated by D. The receptive field size is 9 timesteps.

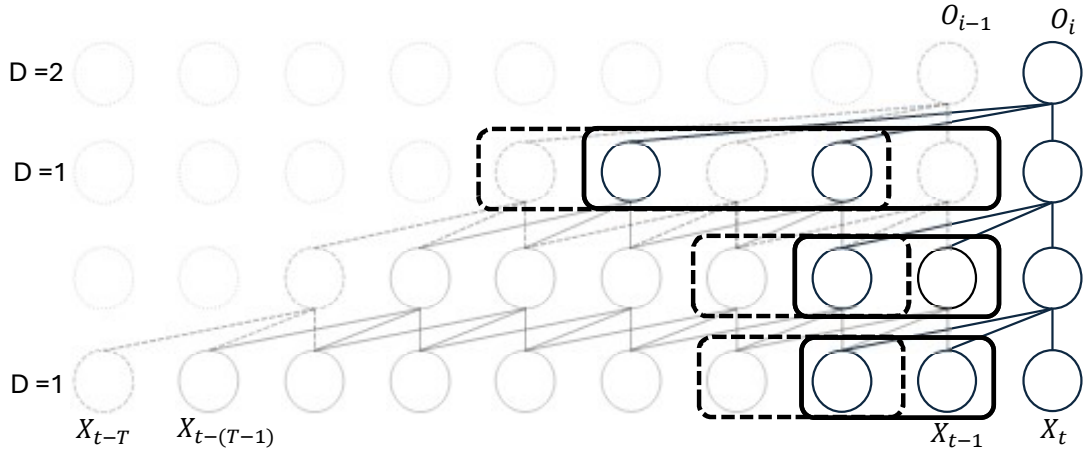


Figure 7.3: Streaming TCN. Dashed black lines represent buffers at timestep $t-1$. Updated buffers (black lines) containing the intermediate outputs at $t-1$ are utilized as input at timestep t.

required for model deployment. By operating on ONNX, a universal model representation compatible with multiple frameworks and devices, the proposed approach ensures portability across software and hardware platforms.

7.1.1 Streaming

7.1.1.1 Convolution Layers

Fig. 7.2 illustrates the execution flow of a TCN in real-time mode. A subset of the convolution operations required to compute the current output value (o_t) were already performed for the previous one (o_{t-1}), as indicated by the dashed black lines. The most recent T inputs, where T represents the network receptive field, can be stored in memory, and the necessary convolution operations can be recomputed [48]. However, real-time inference in resource-constrained environments requires an optimized pipeline.

Fig. 7.3 illustrates the main concept of the streaming approach. Each layer is associated with an independent buffer that stores a history of its inputs (outputs of the previous layer) necessary for processing the next timesteps. The buffer size is given by $(k-1) \times d$, where k

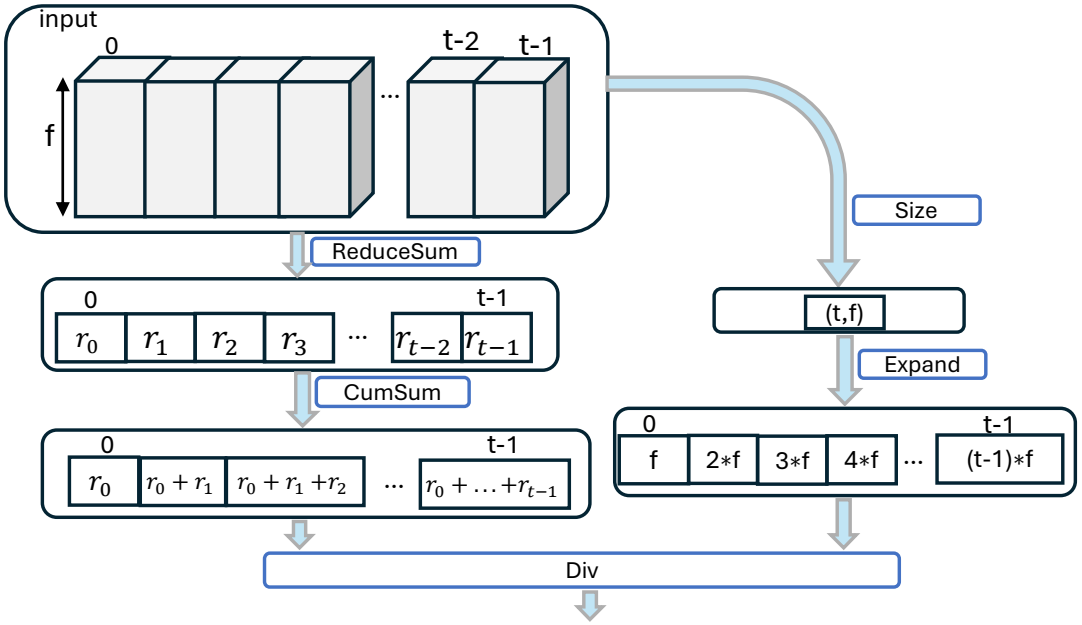


Figure 7.4: Real-time cLN process. `ReduceSum`, `CumSum`, and `Expand` normalize inputs by incorporating previous feature statistics.

represents the kernel size and d denotes the dilation rate of the convolution layer. The algorithm executes the required convolution operations across all layers, updates the corresponding buffers, and generates the network output. Each new input triggers at most one convolution operation per layer. [48]. Each buffer is updated by shifting its contents to remove the oldest values.

As a further enhancement, we add support for multi-timestep streaming. This feature enables concurrent processing of multiple timesteps and provides flexibility to handle varying numbers of timesteps in streaming mode, thereby supporting the "variable length inputs" feature. In this way, the model optimizes computational efficiency, leveraging available hardware resources more effectively. As the number of timesteps increases, the computational demand grows because more data are processed in each pass. However, the overall computational complexity scales sublinearly with the number of timesteps because the model parameters are loaded only once per pass across all timesteps. As a result, total computations increase without a proportional increase in cycles, enhancing overall efficiency and reducing per-timestep processing time. In particular, as more timesteps are processed concurrently, the overall inference time per iteration increases. This flexibility allows users to dynamically adjust the number of timesteps processed per pass, providing a trade-off between computational complexity and latency and enhancing its adaptability to different resource environments and real-time application requirements.

7.1.1.2 cLN

During the calculation of the cumulative statistics for normalization, a `ReduceSum` operation is applied across the feature dimension of the input to the cLN layer, as shown in Fig. 7.4. If the input to the cLN layer has dimensions (t, f), where t represents the receptive field of that layer (equals to t timesteps) and f represents the feature size, the output of the `ReduceSum` operation has the shape $(1, t)$. This reduction operation sums the values along the feature axis to compute the normalization statistics.

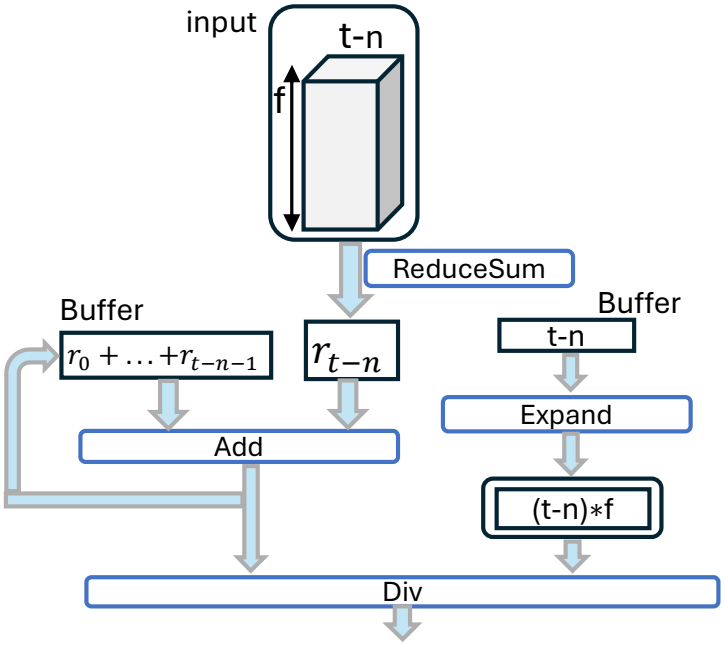


Figure 7.5: Streaming cLN at timestep $t-n$. A buffer stores the last `CumSum` outputs, another one tracks the number of previous timesteps for `Expand`.

Following the `ReduceSum`, a `CumSum` node computes the cumulative sum of these reduced values across the time dimension. Subsequently, each element of the resulting vector must be divided by the number of previous features corresponding to that element. This operation ensures that the cumulative mean and variance are correctly scaled based on the number of features processed up to that point. The `Expand` node plays a crucial role in this step by broadcasting these statistics across the feature dimension, allowing the correct division for each element based on the number of prior input features.

Fig. 7.5 illustrates the implementation of cLN in streaming mode, where the network processes the input at timestep $t-n$. The `ReduceSum` operation is applied to the current timestep. However, to compute the `CumSum` for this timestep, the cumulative sum of all previous timesteps $(r_0+r_1+\cdots+r_{t-n-1})$ is required. A buffer is introduced to store the cumulative sum from the previous timestep, allowing the network to efficiently load the buffer containing $(r_0+r_1+\cdots+r_{t-n-1})$ and add it to the result of `ReduceSum` for the current timestep $t-n$. Additionally, a buffer and a counter track the number of previously processed features for the current timestep. This information is crucial for the `Expand` node, which utilizes the buffered value to ensure correct scaling.

7.1.2 Streaming Transformation Flow

Fig. 7.6 illustrates the modification required to convert a TCN into a streaming network. The first step is to adjust the paths between the input and the first layer, removing any padding initially used to ensure the input size is a multiple of the kernel size and to guarantee the network processes streaming data without unnecessary operations designed for batch processing.

Next, the convolution and cLN layers must be modified. For each convolution layer, an additional input is added to the ONNX graph to load the necessary outputs from previous convolution operations (stored in external buffers). Similarly, an additional output is added to save the current intermediate results of the convolution layer. The values required to process the current timestep(s) determine the sizes. For instance, if the timestep is 1 and the kernel

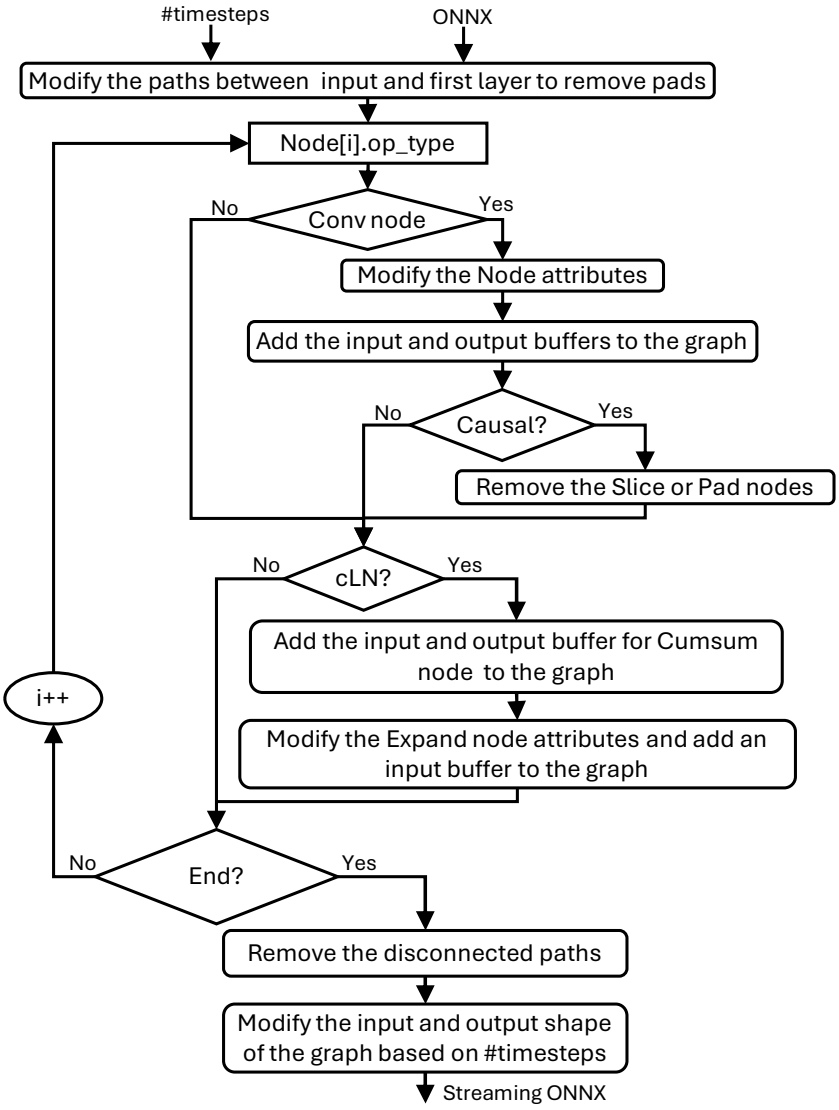


Figure 7.6: Algorithm to convert a trained TCN into a streaming network.

size is k, the convolution layer requires k-1 values from the buffer, with a gap of d (the dilation rate) between them, while the current timestep in the network provides the final value needed to complete the convolution. Fig. 7.3 shows the details of this process. The gap between the inputs of each convolution layer is adjusted based on the number of timesteps, and the dilation rate is updated accordingly.

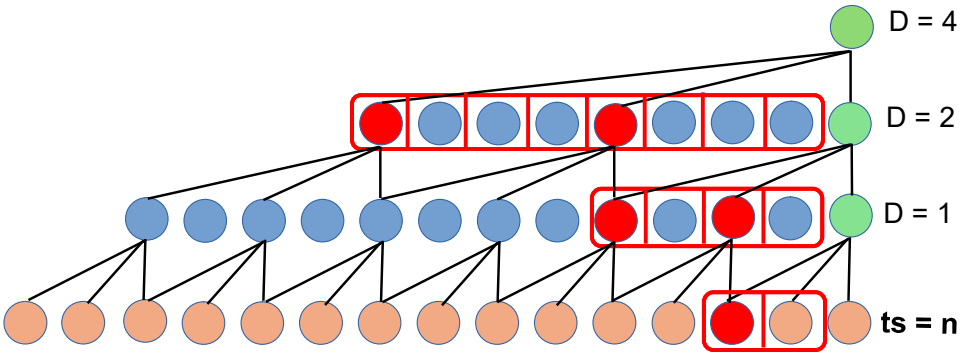
In causal convolutions, the output at time t is only convolved with elements from time t and earlier in the previous layer, preventing future information from being used [10]. To achieve this, left-side padding is typically added. In some implementations, padding may be applied to both input sides for technical reasons. However, the right-side padding is not required for causal convolutions, and a `Slice` node is used to remove the unnecessary outputs generated due to this padding on the right side[2]. In streaming mode, padding and slicing are redundant because the layer already receives the necessary inputs. Since the model does not require additional padding, both the `Pad` and `Slice` nodes can be safely removed, avoiding unnecessary computations and memory usage.

For each cLN layer, corresponding buffers are introduced to enable its operation in stream-

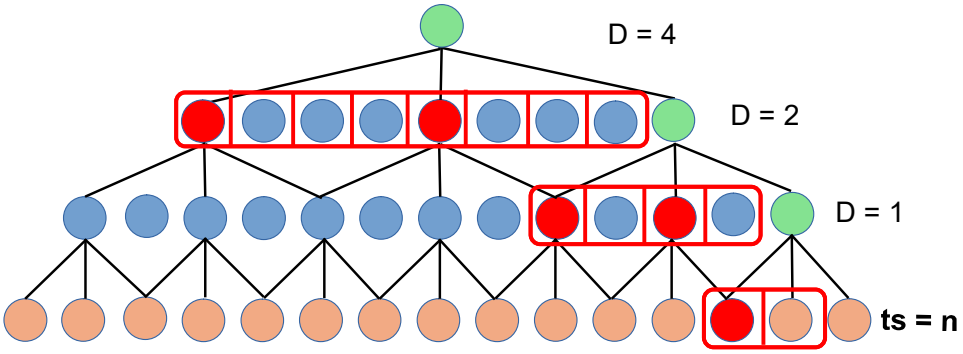
[2] https://discuss.pytorch.org/t/causal-convolution/3456

ing mode. Removing unnecessary nodes may result in disconnected chains because specific nodes (e.g., padding operations) are included in the original network to handle batch processing. For instance, padding nodes that adjust the input shape based on constants may remain after removing the padding itself. Finally, the network's input and output shapes are modified to support processing one or multiple data points at a time. The transformation is computationally efficient and low-latency, with scalability across depth and generalizability to diverse TCN architectures.

To facilitate the developer's tasks, we have provided a comprehensive programming interface that supports creating, initializing, updating, and retrieving buffers based on the network configuration during runtime and executing the streaming network with varying input sizes. Since multi-timestep streaming requires adapting buffer sizes and algorithms to process multiple timesteps concurrently, the tool modifies and reconstructs the network to support multi-timestep processing. Finally, the tool can convert a non-streaming model to a streaming model for both causal and non-causal configurations, ensuring compatibility across various network types. As demonstrated in Figure 7.7a and Figure 7.7b, all the required intermediate convolution results needed to generate the output are present in the buffers (indicated by red circles). Notably, our proposed methodology does not impact accuracy, ensuring that the accuracy metrics remain consistent with those of the non-streaming model.



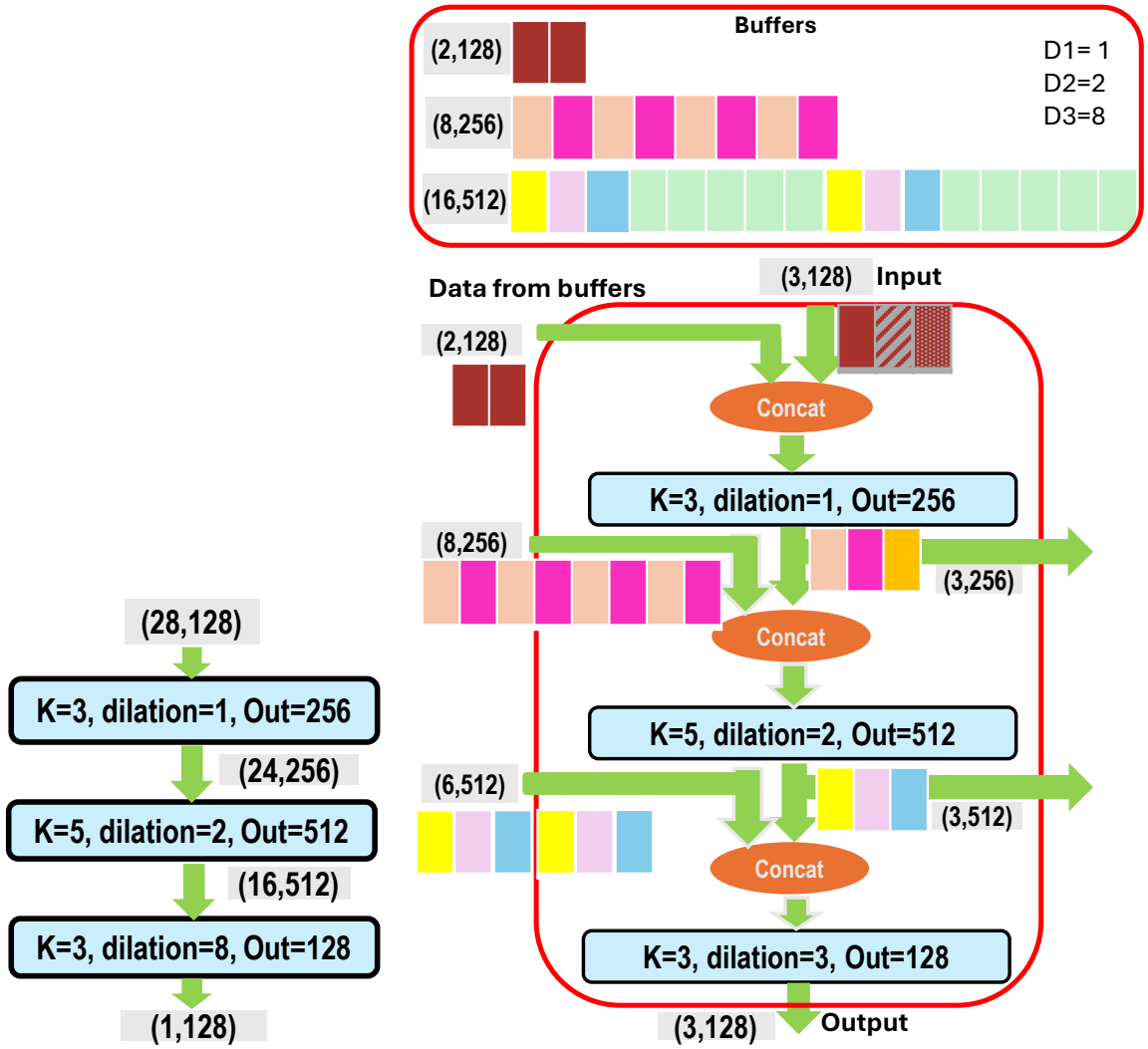
(a) Streaming causal TCN



(b) Streaming non-causal TCN

Figure 7.7: Buffers are indicated by red rectangles, and $ts = n$ is the new timestep. The green convolution operations are computed for this timestep.

Figure 7.8a illustrates an example of a non-streaming TCN consisting of three layers with a receptive field spanning 28 timesteps; in this setup, the network processes the entire receptive field to generate one timestep in the output. The objective is to transform this non-streaming network into a streaming architecture with three timesteps $(ts = 3)$. Initially, our tool iterates through the network to allocate a buffer for each layer (Figure 7.8b top). Afterward, since only the required data (not the entire receptive field of the layer), which is determined by the kernel



(a) (b)

Figure 7.8: (a) Non-streaming network with receptive field of 28 timesteps. (b) The streaming network with three input timesteps. The data for processing each timestep is indicated using consistent colors. Since the dilation rate is one (two) in the first (second) layer, there are overlaps between the data for the last two (one) timesteps. These overlaps are not color-coded.

size (k) and dilation rate (d), is fed to each layer in the streaming mode, the spacing between subsequent inputs supplied to each convolution operation is set (Equation 7.1).

$$d' = \begin{cases} d & \text{if } ts > d \\ ts & \text{otherwise} \end{cases} \tag{7.1}$$

Finally, using the new dilation rate, the tool calculates the amount of data required from each buffer to process each layer, as defined in Equation 7.2, and uses it as the layer's input.

$$input\_buffer = (k-1) \times d' \tag{7.2}$$

7.2 Experimental Results

7.2.1 Experimental Setup

Tinydenoiser [103] introduces an optimized approach for designing and deploying Speech Enhancement (SE) algorithms based on RNNs on low-power MCUs. It employs an optimized software pipeline that interleaves parallel computations of LSTM or GRU blocks using vectorized 8-bit integer (INT8) and 16-bit floating-point (FP16) units, along with manually managed memory transfers of model parameters. We replicated the same setup as the Tinydenoiser [103]. For training, a weighted combination of L1 and Short-Time Fourier Transform (STFT) losses was used as the objective function. The ADAM optimizer was used with a learning rate of 3e-4, and a batch size of 64 was used for 200 epochs. The training uses the Valentini dataset [128], which contains clean and noisy speech audio clips from 28 speakers sampled at 16 kHz. The Perceptual Evaluation of Speech Quality (PESQ) [97] quantitative metric is employed.

Notably, while the chosen Conv-TasNet configuration achieves PESQ scores comparable to Tinydenoiser's and a parameter count similar to Tinydenoiser's, alternative configurations could have been employed to achieve higher PESQ performance. However, the primary objective of this tool is not to maximize PESQ scores but rather to demonstrate the practicality of deploying TCN-based models, such as Conv-TasNet, for real-time streaming applications on resource-constrained devices. We used the 240 MHz frequency and 0.65 V frequency setting for GAP9 to compare energy efficiency with Tinydenoiser. The Post-Training Quantization (PTQ) procedure is included in the GAPflow toolset[3]. We applied the methodology from [42] for INT8 symmetric quantization. Our quantization module imports a trained full-precision model and quantizes it to BF16, FP16, INT8, or a mixed BF16-INT8 format before generating the application code for deployment. Four randomly selected audio files from the validation set are used to calibrate the quantization ranges.

The audio input is sampled at 16 kHz, with STFT features extracted over a 25 ms audio frame following Hanning windowing. For each audio frame, the model takes 257 STFT magnitude values as input and returns 257 gain values. A hop size of 6.25 ms (25% of the window length) is used, defining the real-time constraints for inference. The filtered frequency spectrum is obtained by masking the noisy spectrum, and the result is converted back to the time domain using inverse STFT. Latency can be calculated using (7.3):

$$\begin{aligned}\text{Latency} = &\left(\frac{\text{frame length}}{\text{hop size}}\right) \times \text{hop size} + \text{inference time} \\ &+ (ts - 1) \times \text{hop size}\end{aligned} \tag{7.3}$$

In real-time streaming, the denoised signal is reconstructed using overlap-and-add on the last 4 overlapping predicted windows, with a frame length of 400 samples and a hop size of 100 samples. For our specific setup (frame length = 25 ms, hop size = 6.25 ms), the latency is calculated as 4×6.25 ms + inference time + $(ts - 1) \times 6.25$ ms.

The ONNX runtime version used during the experiments was 1.17.0. For the CPU experiments, we used an 11th Gen Intel(R) Core(TM) i7-11370H processor running at 3.30 GHz.Moreover, a software tool (GAPflow) enables the deployment of NN onto the GAP9.

[3] https://github.com/GreenWaves-Technologies

Table 7.1: PESQ and memory footprint of Tinydenoiser and TCN denoiser after PTQ using FP16, and Mixed INT8-(B)FP16.

	Tinydenoiser						
	LSTM256		**GRU256**		**TCN**		
Quant	**PESQ**	**MB**	**PESQ**	**MB**	**PESQ**	**MB**	**Buffer Size (MB)**
FP32	2.78	4.75	2.78	3.76	**3.02**	3.33	0.096
FP16	2.78	2.37	2.78	1.88	NaN	1.67	0.048
INT8-(B)FP16	2.73	1.37	2.72	1.13	2.98	**0.88**	**0.024**

7.2.2 Accuracy Results

Table 7.1 reports the PESQ scores and memory footprint for the quantized Tinydenoiser models and the TCN denoiser across various quantization configurations. INT8-FP16 quantization is applied in Tinydenoiser, and INT8-BF16 in the TCN denoiser. In the mixed quantization scheme, the first datatype is applied to the RNN (or Conv) layers in Tinydenoiser (or TCN denoiser), while the second datatype is used for the remaining layers. First, the TCN model achieves higher PESQ scores than the Tinydenoiser variants (LSTM256 and GRU256) under the evaluated configurations.

Second, the FP16 configuration results in a not-a-number (NaN) output for the TCN model due to systematic numerical overflow in the cLN layers. To address this issue, we apply FP16 to the convolutional layers and BF16 to the cLN layers to increase the dynamic range, yielding PESQ scores equivalent to the FP32 baseline. Using BF16 across all layers reduces memory usage by 50% without compromising performance. Third, the TCN model can be further optimized by quantizing the convolutional layers to INT8 while maintaining BF16 for the cLN layers. This mixed-precision quantization achieves a 4× reduction in memory usage with minimal impact on PESQ.

7.2.3 Hardware Mapping

7.2.3.1 CPU

In the non-streaming mode, the CPU average inference time per frame is 5.284 ms. This higher computational cost is due to the full processing required at each timestep. In contrast, the streaming network significantly reduces inference time. When the model processes a single timestep (ts=1), the inference time decreases to 0.939 ms. As the number of timesteps increases, the inference time rises but remains considerably lower than that of the non-streaming setup. The inference time for ts=2 is 1.052 ms, increasing to 1.115 ms for ts=3 and reaching 1.186 ms for ts=4.

7.2.3.2 GAP9

Table 7.2 presents the MAC operations and cycles for both the non-streaming and the streaming network with one timestep (ts=1), which serve as baseline references. In the non-streaming setup, the inference time is 113.8 ms for the BF16 quantization and 57.4 ms for the INT8-BF16 quantization. These values exceed the real-time requirement of 6.25 ms per frame. In

Table 7.2: Performance of the non-streaming (NS) and streaming models with different timesteps and quantization on GAP9. The table reports MAC operations (millions), cycles (millions), MAC/Cycle ratio, inference time (ms), and active time (%).

	BF16					**INT8-BF16**				
	MAC	**Cyc**	**MAC/ Cyc**	**inf-time**	**active time**	**MAC**	**Cyc**	**MAC/ Cyc**	**inf-time**	**active time**
NS	92.45	27.31	3.39	113.8	1820	92.05	13.79	6.67	57.48	919.6
ts=1	**0.84**	0.78	1.08	3.2	51.6	0.85	**0.5**	1.7	**2.07**	33.13
ts=2	1.7	1.26	1.35	5.2	41.9	1.74	0.81	2.16	3.35	26.85
ts=3	2.55	1.48	1.73	6.15	32.84	2.6	0.88	2.95	3.67	19.6
ts=4	3.4	1.58	2.15	6.58	26.34	3.47	1.05	**3.3**	4.38	**17.55**

Table 7.3: Comparison with Speech Enhancement Solutions on MCUs.

Model	**Mpar**	**Quant**	**Device**	**inf-time (ms)**	**MAC/ cycle**	**avg power(mW)**	**PESQ**
Tiny LSTM [24]	0.33	INT8	STM32 F746VE	4.26	0.36	143.75	-
Tiny LSTM [24]	0.46			2.39		206.4	-
RNNoise [129]	0.21	INT8	STM32 L476	3.28	0.45	-	-
Tiny denoiser [103]	1.24	INT8-FP16	GAP9	2.5	2.11	10.94	2.73
Tiny denoiser [103]	0.98			**1.7**	2.41	7.44	2.72
ts=1 (ours)	0.83	INT8-BF16	GAP9	2.07	1.7	9.06	**2.98**
ts=4 (ours)	0.83			4.3	**3.3**	**4.80**	**2.98**

contrast, the streaming network with ts=1 achieves an inference time of only 2.07 ms (INT8-BF16).

Table 7.2 also shows substantial reductions in computational load with the streaming configuration. In the ts=1 setting, the MAC operations are reduced by 108.9× and the number of cycles by 27.7× compared to the non-streaming configuration. For the INT8-BF16 configuration, the reported MACs include additional operations due to conversions between INT8 and BF16 formats, resulting in a slightly higher count. In this context, MACs denote multiply–accumulate operations and are used as a proxy for computational cost.

Furthermore, adopting multi-timestep streaming increases the MAC/Cycle ratio. For ts=1, the network processes only one timestep at a time, yielding a lower MAC/Cycle (1.7) ratio but minimal latency; with ts=4, the MAC/Cycle ratio improves to 3.3, while the inference time is 4.38 ms, which is still within the acceptable range for real-time processing (25 ms).

Table 7.2 shows a significant reduction in active time percentage as the number of timesteps (ts) increases. With ts=1, the active time is 51.6% for BF16 and 33.13% for INT8-BF16, decreasing to 26.34% and 17.55%, respectively, at ts=4. This reduction in active time corresponds to a lower duty cycle.

Table 7.4: Comparison of different configurations based on the Conv-TasNet architecture. R represents the number of repetitions. X denotes the number of convolutional blocks within each repetition. D is the maximum dilation rate. RF indicates the receptive field measured in seconds. Model sizes are expressed in million parameters.

Config	R	X	D	RF(s)	Model size	Baseline Cycles(M)	Baseline MACs(M)
1	2	5	16	0.31	1.3	16.6	86.4
2	6	4	8	0.45	2.66	48.6	245.7
3	2	6	32	0.63	1.45	41.2	211
4	6	5	16	0.93	3.2	131.8	632.4
5	7	5	16	1.09	3.77	179.5	846.7
6	8	5	16	1.24	4.28	229.1	1092.1
7	4	6	32	1.27	2.66	148.9	725.4
8	2	7	64	1.28	1.65	104	498.6
9	2	8	128	2.55	1.85	236.4	1151.1
10	3	8	128	3.83	2.66	552.6	2340.7

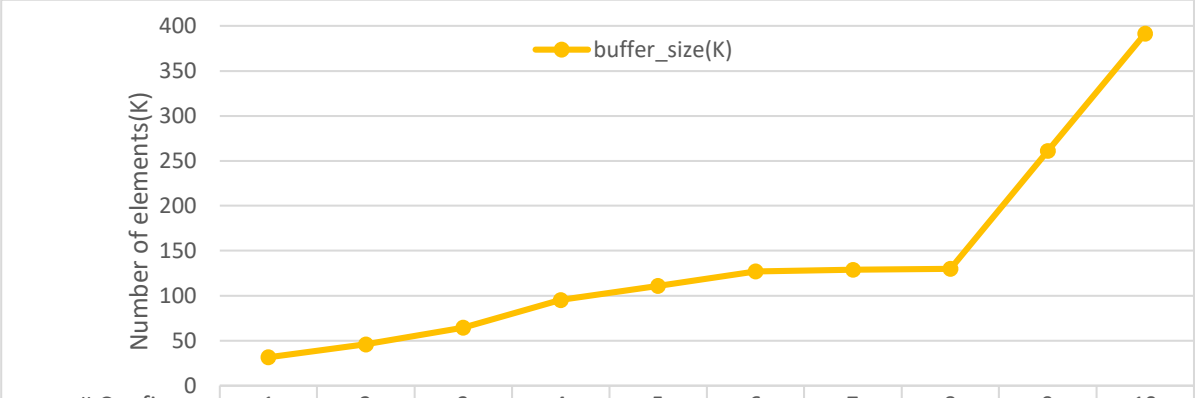
7.2.4 Comparison with Speech Enhancement solutions on MCUs

Table 7.3 compares our solution and other SE models designed for MCUs. At ts=1, our solution achieves an inference time of 2.07 ms, which surpasses RNNoise's (3.28 ms) and is comparable to TinyLSTM's (2.39 ms). Our approach achieves a MAC/Cycle ratio of 1.7 at ts=1, which is competitive compared to other models, such as TinyLSTM and RNNoise, with ratios of 0.36 and 0.45, respectively. By increasing the timesteps to 4 (ts=4), our model achieves a superior MAC/Cycle ratio of 3.3. Despite the higher inference time, the latency still meets real-time processing requirements.

Regarding power consumption, our model consumes 9.06 mW at ts=1, which is considerably lower than TinyLSTM. At ts=4, the power consumption decreases to 4.80 mW. Using multi-timestep processing, designers can save power to extend the battery lifetime or perform additional tasks within the same power budget. This comparison demonstrates that multi-timestep streaming provides a trade-off between computational (and energy) efficiency and latency. Increasing the timesteps enhances the MAC/Cycle ratio, reducing the relative active time in the processed window at the cost of higher overall system latency. This flexibility allows for a dynamic balance between real-time performance and computational requirements, making it adaptable to various application demands.

7.3 Network Architecture: Design Choices and Trade-offs

We evaluate our library by comparing various configurations of the Conv-TasNet model, and assess their efficiency in terms of buffer sizes, the reduction rate (RR) of MACs, and cycles of the streaming model compared to the non-streaming one, and MAC per cycle for processing 1, 2, and 3 timesteps. The sample rate is 8k. We do not report accuracy values since our methodology does not affect them: the accuracy metrics remain consistent with those reported in the Conv-TasNet paper [67]. We provide real-world hardware evaluation using the GAP9



# Config	1	2	3	4	5	6	7	8	9	10
buffer_size(K)	31.744	46.08	64.512	95.232	111.104	126.976	129.024	130.048	261.12	391.68
Receptive field(s)	0.31	0.45	0.63	0.93	1.09	1.24	1.27	1.28	2.55	3.83

Figure 7.9: Effect of varying network configuration in Conv-TasNet on buffer size for streaming.

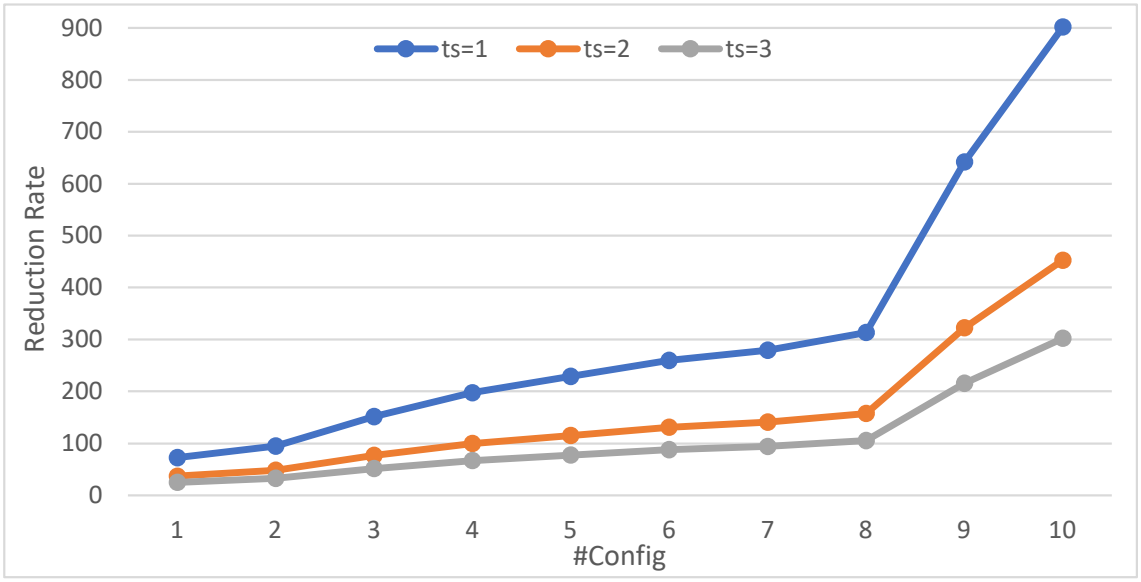
AI chip.

To explore the consistency of receptive field sizes across diverse network configurations with different values of R (deeper network architecture) and X (larger dilation rate), we examined several architectures as detailed in Table 7.4. For instance, configurations 6, 7, and 8 maintain a receptive field of approximately 1.2 seconds but with different R and X values, resulting in varying model sizes. Configuration 6 has 4.28 million parameters, whereas configuration 8 has 1.65 million. These variations demonstrate our library's flexibility to explore different network architectures before the training phase, which is particularly important for deploying TCN models on edge devices such as the GAP9 AI chip, where memory is limited and necessitates a balance between receptive field size and buffer memory for practical deployment.

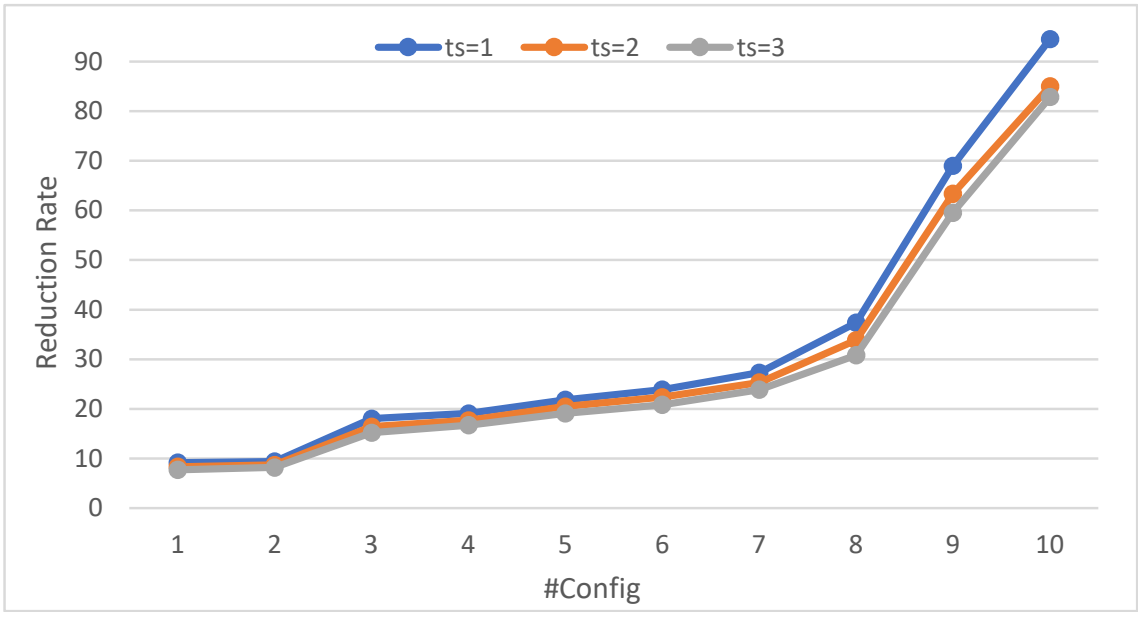
Figure 7.9 illustrates the buffer size requirements for each Conv-TasNet configuration concerning their respective receptive fields. These parameters influence the network's receptive field. Intuitively, a larger receptive field enhances performance (emphasizing the importance of capturing temporal dependencies) [67]; however, it requires substantial memory.

Figures 7.10a and 7.10b show a comprehensive evaluation of MACs and cycle reduction rates for the ConvTasNet models. The reduction rate is calculated by dividing the metric value of the non-streaming model by that of the streaming model for a fixed number of timesteps. The observed trend indicates that as the receptive field increases, the reduction rate also becomes higher due to the increased occurrence of repetitive computations that can be avoided with our method. For instance, with a receptive field of 0.31 seconds (2480 audio samples), the reduction rate of cycles and MACs is $9.1\times$ and $72.6\times$, respectively. As the receptive field is extended to 3.8 seconds (30.4k audio samples), these reduction rates increase significantly to $94.5\times$ and $901\times$ for cycles and MACs, respectively.

While the streaming model achieves considerable efficiency advantages for a single timestep, the reduction rate, particularly for MACs, decreases as additional timesteps are processed. As shown in Figure 7.11a and 7.11b, the non-streaming model experiences only a marginal increase in computations. For example, the number of MACs for the non-streaming model increases from 86.4 million with one timestep to 87.6 million with two timesteps. However, as illustrated in Figures 7.11c and 7.11d, when the number of timesteps increases to two, the input size doubles in the streaming mode, causing the number of MACs to rise from 1.1 million



(a) MAC reduction rate



(b) Cycle reduction rate

Figure 7.10: Comparison of reduction rates across different numbers of timesteps in streaming mode compared to non-streaming mode.

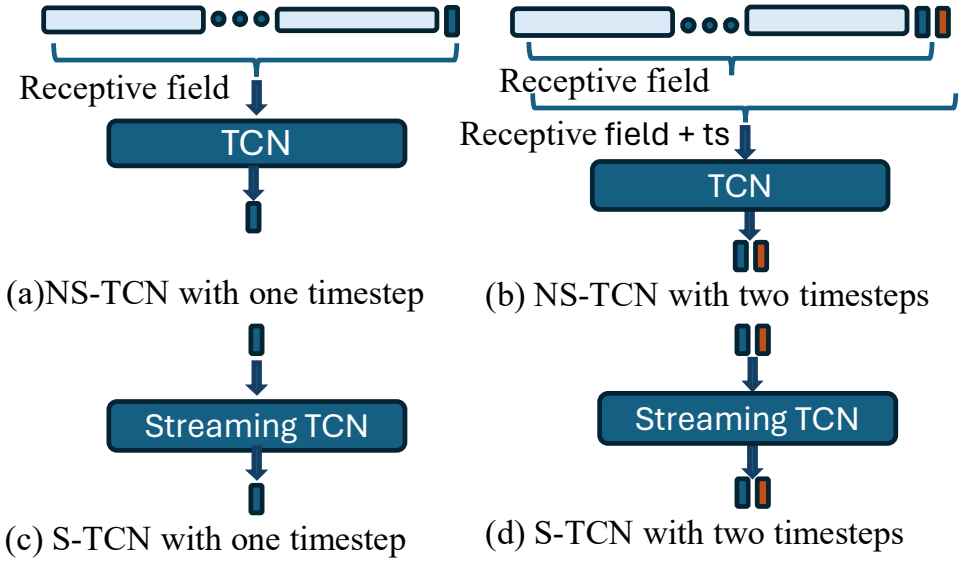


Figure 7.11: Comparison of multi-timesteps approach in non-streaming TCN (NS-TCN) and streaming modes TCN (S-TCN).

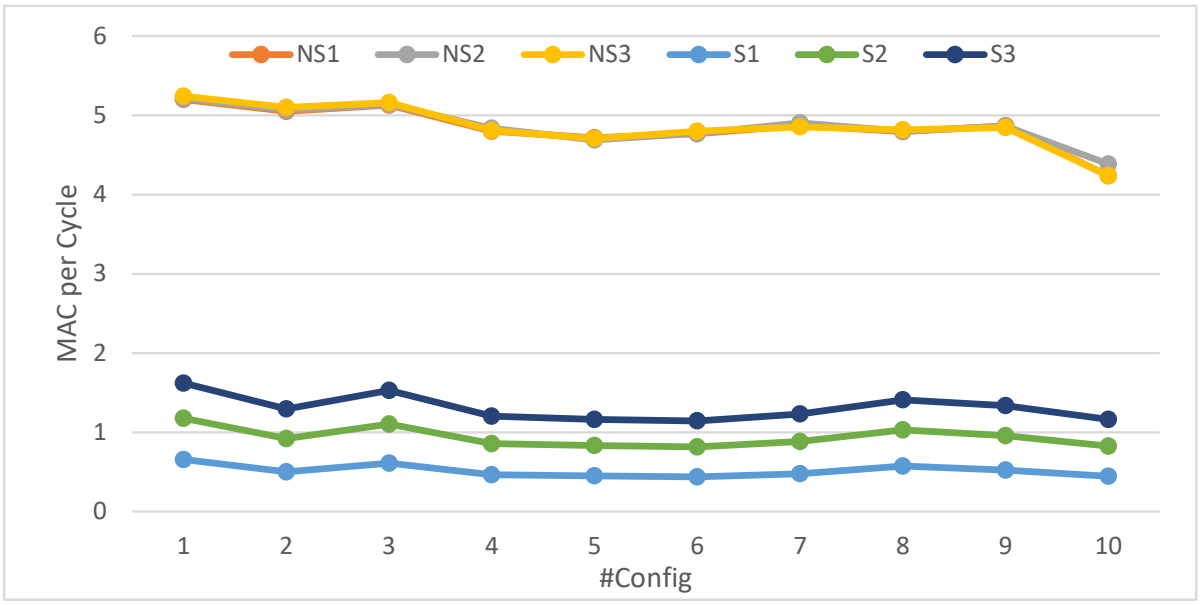


Figure 7.12: MAC per cycle for different network configurations. NS and S represent non-streaming and streaming modes, respectively.

to 2.3 million.

Although the reduction rate decreases in this scenario, it is important to note that the number of MACs is still significantly lower in the streaming model (2.3 million) compared to the non-streaming model (87.6 million). The decrease in cycles with additional timesteps is not as pronounced as in MACs due to the memory hierarchy's network loading phase, which occurs once regardless of the number of timesteps processed. This implies that the cycle increase is relatively small compared to the overhead associated with data movement between memory hierarchies. However, waiting for additional timesteps in non-streaming systems is incompatible with the constraints of real-time applications due to increased network latency.

Figure 7.12 explores the MACs per cycle ratio, demonstrating that the MACs per cycle in the streaming model increase with the addition of more timesteps. Unlike the non-streaming model, which exhibits consistent efficiency, the performance of the streaming model is variable. This variability underscores the streaming model's robust capability to meet real-time processing demands.

7.4 Conclusion

This chapter introduces a Python library that automates converting non-streaming TCN models into their streaming counterparts, thereby significantly reducing deployment time and computational resource requirements. By enabling multi-timestep streaming, our tool balances network versatility and computational demands. Our evaluation demonstrates the effectiveness of our library in reducing computational overhead while maintaining accuracy, making TCN-based models more accessible for real-time applications in resource-constrained environments. Additionally, we provide real-world hardware evaluation using the GAP9 AI chip from GreenWaves Technologies, showcasing the practical feasibility of deploying TCN-based models on edge devices.

CHAPTER

8

CONCLUSION & FUTURE WORK

This thesis has explored how **transprecision computing**—the systematic tuning and mixing of numeric precisions across software and hardware layers—can enable energy-efficient, real-time DSP and ML on resource-constrained MCU devices. By targeting floating-point formats smaller than 32 bits, including FP16, bfloat16, and FP8 variants, we demonstrated how precision can be treated not as a fixed design choice but as a controllable dimension of performance, accuracy, and energy efficiency.

The thesis began by quantifying the behavior of **SmallFloat formats** beyond their traditional deep learning domain. Through detailed experimentation, we showed how the correct choice of exponent–mantissa balance and scaling policy can maintain accuracy while drastically reducing memory footprint and computation cost. We then introduced **TransLib**, an open-source transprecision kernel library that standardizes precision exploration across a suite of DSP and ML workloads. TransLib provides a unified platform for analyzing accuracy–efficiency trade-offs, integrating Python-based golden models and C implementations optimized for multi-core PULP-class architectures.

As a case study, we proposed an optimized **Self-Organizing Map** for bacterial genome identification on the PULP platform, demonstrating how architectural mapping (horizontal vs. vertical), tiling, and SIMD vectorization interact to achieve efficient execution under strict memory constraints. Together, these contributions establish a reproducible methodology and a set of practical tools for applying transprecision principles to real-world, embedded workloads.

The second major contribution, **StreamEase**, addressed the deployment of TCNs on ultra-low-power devices. It automated the transformation of offline-trained models into **stream-oriented pipelines** that sustain real-time operation without retraining or accuracy degradation. This framework bridges the gap between static deep learning models and dynamic embedded inference, opening new opportunities for continuous bio-signal monitoring and time-series analytics. To validate its effectiveness, StreamEase was evaluated through a **speech enhancement case study** deployed on the **GAP9** platform, demonstrating its ability to maintain inference accuracy while meeting strict latency and memory constraints characteristic of MCU-class systems.

At a higher level, three key insights emerged from this research:

- **No single optimization dominates** – mixed-precision computation and streaming approaches outperform homogeneous reductions when operand sensitivity and memory pressure vary across stages.
- **FP8 is viable beyond deep learning** – with proper scaling and accumulation in FP16/FP32, FP8 can preserve accuracy while delivering significant energy and bandwidth savings.
- **Hardware–software co-design is essential** – platform features such as SIMD width, on-chip scratchpad memory, and DMA tiling policies define the achievable Pareto frontier between accuracy, latency, and energy.

Looking ahead, several directions naturally extend this thesis. Future research should aim to broaden the FP8 and transprecision methodologies by incorporating automated scaling calibration and additional floating-point variants. Automated **mixed-precision search and scheduling** can further optimize trade-offs among accuracy, latency, and memory within given energy budgets. At the framework level, StreamEase can evolve into a **compiler-style pass** supporting other causal and attention-based models, equipped with static performance analysis and multi-target code generation. Finally, **improving tooling, verification, and reproducibility**—through continuous integration, differential testing, and numerical robustness evaluation—will help turn transprecision computing from a research technique into a deployable engineering discipline.

In conclusion, this thesis demonstrates that principled transprecision design, supported by reproducible open-source tools, can transform low-power embedded platforms from simple sensor nodes into intelligent, real-time analytic engines. By uniting format exploration, streaming transformation, and reproducibility, this thesis lays the foundation for a new generation of ultra-efficient, precision-adaptive computing for time-series and bio-signal processing at the edge.

APPENDIX

A

SUPPLEMENTARY MATERIAL: PUBLIC CODE REPOSITORIES AND OPTIMIZATION OVERVIEW

This appendix, first, catalogues the public code repositories that underpin our experiments and ensure full reproducibility across targets. We then synthesize the optimization strategies explored over multiple kernels and workloads, highlighting when and why each technique is most effective on PULP-class architectures (Table 8.1).

A.1 Public Code Repositories

This section provides implementation details and reproducibility information for the software frameworks and libraries developed throughout this thesis. These resources ensure that all experiments and performance evaluations can be independently reproduced.

A.1.1 TransLib – A Transprecision Kernel Library for IoT End-Nodes

Project Name: TransLib: A Library to Explore Transprecision Floating-Point Arithmetic on Multi-Core IoT End-Nodes
Repository URL: https://github.com/ahmad-mirsalari/TransLib
License: Apache 2.0

A.1.1.1 Overview

TransLib is a modular kernel library developed to explore *transprecision computing techniques*. It enables systematic evaluation of multiple floating-point formats—including `FP32`, `FP16`, `FP16ALT`, `bfloat16`, and `float8` variants—in terms of accuracy, memory footprint, and computational cost. The library targets the embedded MCUs; it uses **PULP** and

GAP9 as example platforms to demonstrate portability and performance. TransLib supports both *fixed-precision* and *mixed-precision* arithmetic.

A.1.1.2 Structure and Features

Each kernel is self-contained and includes:

- A **Python-based golden model** for dataset generation, functional checking, and precision benchmarking.
- A **C-based implementation** optimized for execution on embedded systems.

The repository is structured into:

- `fixed_precision/` – Kernels using a uniform precision format (e.g., `FP32`, `FP16`)
- `mixed_precision/` – Kernels supporting mixed-precision assignments (e.g., inputs, weights, intermediate results, outputs)
- `utils/` – Common helpers (format conversion, rounding/denormals, and `mpmath`-based high-precision checks).

A.1.1.3 Implemented Kernels

- Matrix Multiplication
- Convolution
- Discrete Wavelet Transform (DWT)
- Fast Fourier Transform (FFT)
- Finite Impulse Response (FIR) Filtering
- K-Means Clustering
- Support Vector Machine (SVM) Classification

A.1.1.4 Benchmarks / Datasets

These kernels are applicable in various signal processing and machine learning scenarios such as:

- Low-power embedded inference (e.g., physiological monitoring)
- Real-time signal processing on edge devices
- Feature extraction for unsupervised clustering and classification

A.1.1.5 Reproducibility

All dependencies, setup instructions, and usage workflows are fully documented in the repository's README.md.

The user can:

- Install required Python and toolchain packages
- Generate synthetic or application-specific datasets
- Run golden models for accuracy validation
- Deploy and test kernels on actual hardware using PULP SDK and GVSoC

A.1.2 StreamEase – Streaming Conversion for TCNs on Low-Power MCUs

Project Name: StreamEase: Enabling Real-Time Inference of Temporal Convolutional Networks on Low-Power MCUs with Stream-Oriented Automatic Transformation
Repository URL: https://github.com/ahmad-mirsalari/StreamEase
License: Apache 2.0

A.1.2.1 Overview

StreamEase is a Python-based software toolkit that enables the transformation of TCNs into streaming implementations. The resulting models maintain their original accuracy while being optimized for low-latency, low-memory inference on resource-constrained microcontrollers (MCUs). This automation substantially reduces deployment complexity and supports real-time applications on a wide range of targets, including CPUs, embedded systems, and parallel ultra-low-power (PULP) platforms.

A.1.2.2 Structure and Features

The repository is modular and includes:

- `streamease/` – Core streaming engine: buffering, quantization, scheduling.
- `examples/` – TCN conversion and deployment examples with benchmark support.
- `utils/` – Integration scripts for ONNX Runtime and GreenWaves' `nntool`.
- `setup_env.sh` – Environment setup and `PYTHONPATH` configuration script.
- `requirements.txt` – Python dependency list.

A.1.2.3 Benchmarks / Datasets

The framework is suitable for streaming-based real-time applications such as:

- Human activity recognition and gesture classification
- Biological signal monitoring
- Real-time speech enhancement

A.1.2.4 Key Capabilities

- **Automatic TCN-to-stream conversion** preserving model accuracy.
- **Support for quantized and float-based networks** using ONNX Runtime and `nntool`.
- **Real-time streaming execution** optimized for low-power MCUs.
- **Modular, extensible design** for research and production-ready deployments.

A.1.2.5 Reproducibility

StreamEase is fully open-source and version-controlled. It supports end-to-end automation of the deployment pipeline. All experiments are reproducible via example scripts, and software dependencies are fully specified in `requirements.txt`.

For detailed setup instructions, streaming conversion steps, and code documentation, refer to the project README: https://github.com/ahmad-mirsalari/StreamEase

A.1.3 SOM-on-PULP – Optimizing Self-Organizing Maps for Bacterial Genome Identification

Project Name: Optimizing Self-Organizing Maps for Bacterial Genome Identification on Parallel Ultra-Low-Power Platforms
Repository URL: https://github.com/ahmad-mirsalari/SOM-on-PULP
License: Apache 2.0

A.1.3.1 Overview

This repository implements a SOM kernel optimized for bacterial genome identification on the PULP platform. The implementation features both Python-based modeling for golden reference generation and C-based embedded code tailored for execution on low-power multi-core architectures. The framework supports diverse floating-point configurations—including transprecision and mixed-precision arithmetic—to explore trade-offs between accuracy, performance, and energy efficiency.

A.1.3.2 Structure and Features

- **Python kernel**: Generates input sequences, computes golden outputs, and includes customizable evaluation metrics for accuracy assessment.
- **C implementation**: Designed for PULP architectures with optional support for vector instructions (SIMD) and parallelism across cores.
- **Format support**: Supports `FP32`, `FP16`, `FP16ALT`, `FP8`, and mixed-precision computations.
- **Mapping modes**: Horizontal and vertical mappings are supported to match core count and input size constraints.
- **Toolchain integration**: Compatible with PULP-SDK and GVSoC simulator.

A.1.3.3 Benchmarks / Datasets

The system targets bioinformatics pipelines for genome analysis. Input sequences mimic bacterial DNA slices, which are classified using a trained SOM to infer bacterial class identity. Parameters such as slice length, number of neurons, and number of bacteria can be customized.

A.1.3.4 Reproducibility

All models, toolchains, and configurations are version-controlled and documented for reproducibility. The Python script `data_generator.py` allows for golden model regeneration with customizable FP formats. C implementations are configurable through Makefile flags (e.g., `fmt`, `cores`, `vec`, `IN_ORDER`). For complete setup instructions and usage examples, refer to the repository README.

A.2 Summary on Optimization Strategies

To strengthen the contribution and provide a clearer perspective on the optimization landscape explored in this thesis, we summarize the key techniques applied across multiple kernel implementations in Table A.1. This structured overview explicitly maps each technique to its corresponding kernel, identifies the performance conditions under which it yields optimal results, and outlines the dataset or application scenario where it is most effective.

The comparative summary presented in Table A.1 illustrates that optimization effectiveness is highly dependent on both application context and hardware constraints—there is no universally dominant technique. Each method offers distinct trade-offs, and its suitability is guided by task-specific requirements such as precision, sensitivity, memory footprint, execution latency, and parallel processing capabilities.

For example, homogeneous format reduction, such as adopting FP16 throughout computation, is well-suited to general-purpose workloads with moderate precision demands, particularly when vectorization support is available. In contrast, mixed-precision computation is advantageous in scenarios where different parts of an algorithm can tolerate different levels of precision, enabling improvements in memory efficiency and energy consumption without significantly affecting overall accuracy.

Hardware-level enhancements such as SIMD vectorization and multi-core parallelism further improve throughput, especially in time-critical or signal processing applications. These techniques are particularly beneficial on platforms like PULP, which offer fine-grained control over parallel execution.

To address memory constraints, especially in applications like bacterial genome analysis or streaming inference, tile-based parallelism is employed to segment large tensors, enabling partial loading during computation. Additionally, horizontal and vertical mapping strategies in SOM allow architectural tuning depending on available cores and data layout, offering flexible deployment options.

Finally, the integration of streaming transformation—via the StreamEase—demonstrates a specialized optimization for Temporal Convolutional Networks (TCNs), enabling real-time execution on microcontrollers without retraining or accuracy loss.

Table A.1: Comparative Overview of Optimization Techniques Across Kernels

Technique	**Application Context / Kernel**	**Best Performing Scenario**	**Dataset / Usage Scenario**
Homogeneous FP Format Reduction	All TransLib's (MatMul, FFT, FIR, etc.)	When maintaining a single precision is sufficient and compatible with SIMD operations	General-purpose workloads with mild accuracy constraints
Mixed-Precision Format Selection	MatMul, SOM, FIR, SVM, Conv, K-Means, DWT, TCN	When accuracy can be preserved with lower-precision inputs/weights while reducing memory/energy	Kernels with asymmetric operand sensitivity
SIMD Vectorization	All TransLib's kernels, SOM, TCN	Effective with FP16, FP8, FP16ALT for exploiting PULP SIMD extensions	Real-time signal processing and low-latency applications
Tile-Based Parallelism	SOM, TCN, and other large-tensor kernels	When input/weight tensors exceed memory limits, enables partial loading	Long bacterial sequences, streaming pipelines, or high neuron-count setups.
Multi-core Parallelism	All kernels (MatMul, TCN, SOM, etc.)	Improves throughput via core-level workload partitioning	Large-scale inference with high parallelism potential
Horizontal vs Vertical Mapping	SOM	Horizontal: neuron-major mapping for multi-core execution; Vertical: input-major mapping for fewer cores	Chosen dynamically based on input/weight size and core count
Streaming Transformation (StreamEase)	TCN-based models	Real-time execution on MCUs without retraining, preserving model accuracy	sequence signal analysis, continuous EMG inference, time-series applications

BIBLIOGRAPHY

[1] PULP Software Development Kit. Website: https://github.com/pulp-platform/pulp-sdk. Accessed: 2022-09-05.

[2] Top500: The list and linpack methodology. top500.org. Accessed 2025-10-13.

[3] M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy. Nvidia Hopper Architecture In-Depth, 2022. 2, 4, 6.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK Users' Guide. SIAM, 3rd edition, 1999.

[5] Arm Ltd. Arm cortex-m55 processor. https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m55, 2020.

[6] Arm Ltd. Arm helium technology (m-profile vector extension, mve) — overview. https://www.arm.com/technologies/helium, 2020.

[7] Arm Ltd. Arm architecture reference manual for the armv8-m architecture. https://developer.arm.com/documentation, 2022.

[8] Arm Ltd. Arm ethos-u55 npu: Technical reference manual. https://documentation-service.arm.com/, 2022.

[9] S. Bai, J. Z. Kolter, and V. Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling, 2018.

[10] S. Bai, J. Z. Kolter, and V. Koltun. An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling, 2018.

[11] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. D. Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong. Mlperf tiny benchmark, 2021.

[12] M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. xLSTM: Extended Long Short-Term Memory. arXiv preprint arXiv:2405.04517, 2024.

[13] L. Bertaccini, G. Paulin, M. Cavalcante, T. Fischer, S. Mach, and L. Benini. MiniFloats on RISC-V Cores: ISA Extensions With Mixed-Precision Short Dot Products. IEEE Transactions on Emerging Topics in Computing, 12(4):1040–1055, 2024.

[14] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi. Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In 2021 IEEE 39th International Conference on Computer Design (ICCD), pages 409–416, 2021.

[15] M. Carreras, G. Deriu, L. Raffo, L. Benini, and P. Meloni. Optimizing Temporal Convolutional Network inference on FPGA-based accelerators, 2020.

[16] H. Chen, T. Xiang, K. Chen, and J. Lu. Nonlinear residual echo suppression based on multi-stream conv-tasnet, 2020.

[17] S. Cherubin, D. Cattaneo, M. Chiari, and G. Agosta. Dynamic Precision Autotuning with TAFFO. ACM Trans. Archit. Code Optim., 17(2), 2020.

[18] S. Dustdar, V. C. Pujol, and P. K. Donta. On distributed computing continuum systems. IEEE Transactions on Knowledge and Data Engineering, 35(4):4092–4105, 2023.

[19] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In Proc. of the 38th Annual Int. Symposium on Computer architecture, pages 365–376, 2011.

[20] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In ASPLOS XVII, 2012.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pages 449–460, 2012.

[22] European Commission. Open transprecision computing (oprecomp) — periodic/final reporting. https://cordis.europa.eu/project/id/732631/reporting, 2020. Project reporting page for OPRECOMP (Grant Agreement ID: 732631).

[23] X. Fan. Chapter 1 - introduction to embedded and real-time systems. In X. Fan, editor, Real-Time Embedded Systems, pages 3–13. Newnes, Oxford, 2015.

[24] I. Fedorov, M. Stamenovic, C. Jensen, L.-C. Yang, A. Mandell, Y. Gan, M. Mattina, and P. N. Whatmough. TinyLSTMs: Efficient neural speech enhancement for hearing aids. arXiv preprint arXiv:2005.11138, 2020.

[25] M. Fishman, B. Chmiel, R. Banner, and D. Soudry. Scaling fp8 training to trillion-token llms. arXiv preprint arXiv:2409.12517, 2024.

[26] W. Fornaciari, G. Agosta, D. Cattaneo, L. Denisov, A. Galimberti, G. Magnani, and D. Zoni. Hardware and Software Support for Mixed Precision Computing: a Roadmap for Embedded and HPC Systems. In 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 1–6, 2023.

[27] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini. Pulp-nn: Accelerating quantized neural networks on parallel ultra-low-power risc-v processors. Philosophical Transactions of the Royal Society A, 378(2164):20190155, 2020.

[28] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices, year=2017. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 25(10):2700–2713.

[29] J. S. P. Giraldo, V. Jain, and M. Verhelst. Efficient Execution of Temporal Convolutional Networks for Embedded Keyword Spotting. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 29(12):2220–2228, 2021.

[30] V. Gnanasambandapillai, A. Bayat, and S. Parameswaran. MESGA: An MPSoC based embedded system solution for short read genome alignment. In 2018 23rd Asia and South Pacific Design Automation Conf. (ASP-DAC), pages 52–57.

[31] Google Cloud TPU Team. Improve your model's performance with bfloat16. https://cloud.google.com/tpu/docs/bfloat16, 2025. By default, TPUs multiply in bfloat16 and accumulate in FP32. Accessed: 2025-10-15.

[32] A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. arXiv preprint arXiv:2312.00752, 2023.

[33] A. Gu, K. Goel, and C. Ré. Efficiently modeling long sequences with structured state spaces. arXiv preprint arXiv:2111.00396, 2021.

[34] R. Gupta, V. Mahendran, and V. Badarla. vStream IT: Video Streaming for Resource Constrained IoTs - An Optimal Control Approach. In 2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS), pages 129–134, 2024.

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

[36] J.-Y. He, Z.-Q. Cheng, C. Li, W. Xiang, B. Chen, B. Luo, Y. Geng, and X. Xie. DAMO-StreamNet: Optimizing Streaming Perception in Autonomous Driving. arXiv preprint arXiv:2303.17144, 2023.

[37] N. J. Higham and T. Mary. Mixed precision algorithms in numerical linear algebra. Acta Numerica, 31:347–414, 2022.

[38] S. Hochreiter. Long Short-term Memory. Neural Computation MIT-Press, 1997.

[39] P. Huang, C. Wang, W. Liu, F. Qiao, and F. Lombardi. A hardware/software co-design methodology for adaptive approximate computing in clustering and ann learning. IEEE Open Journal of the Computer Society, 2:38–52, 2021.

[40] C. Ieracitano, N. Mammone, A. Hussain, and F. C. Morabito. A Novel Multi-Modal Machine Learning Based Approach for Automatic Classification of EEG Recordings in Dementia. Neural Netw., 123(C):176–190, mar 2020.

[41] Intel Labs. Fp8 emulation toolkit. https://github.com/IntelLabs/FP8-Emulation-Toolkit, 2022.

[42] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference, 2017.

[43] C. Jannu and S. D. Vanambathina. Multi-stage progressive learning-based speech enhancement using time–frequency attentive squeezed temporal convolutional networks. Circuits, Systems, and Signal Processing, 42(12):7467–7493, 2023.

[44] H. Jasak. Openfoam: Open source cfd in research and industry. International journal of naval architecture and ocean engineering, 1(2):89–94, 2009.

[45] L. Jiao, R. Zhang, F. Liu, S. Yang, B. Hou, L. Li, and X. Tang. New Generation Deep Learning for Video Object Detection: A Survey. IEEE Transactions on Neural Networks and Learning Systems, 33(8):3195–3215, 2022.

[46] B. Johnson. Urban Land Cover. UCI Machine Learning Repository, 2013. DOI: https://doi.org/10.24432/C53S48.

[47] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey. A study of bfloat16 for deep learning training. arXiv preprint arXiv:1905.12322, 2019.

[48] P. Khandelwal, J. MacGlashan, P. Wurman, and P. Stone. Efficient real-time inference in temporal convolution networks. In 2021 IEEE International Conference on Robotics and Automation (ICRA), pages 13489–13495, 2021.

[49] J. Kim, J. Lee, G. Park, B. Kim, S. J. Kwon, D. Lee, and Y. Lee. An investigation of fp8 across accelerators for llm inference. arXiv e-prints, pages arXiv–2502, 2025.

[50] V. Kishore, N. Tiwari, and P. Paramasivam. Improved Speech Enhancement Using TCN with Multiple Encoder-Decoder Layers. In Interspeech, pages 4531–4535, 2020.

[51] T. Kohonen. The self-organizing map. Proc. of the IEEE, 78(9):1464–1480, 1990.

[52] A. Kuzmin, M. V. Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort. Fp8 quantization: The power of the exponent, 2022.

[53] A. Kuzmin, M. Van Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort. FP8 Quantization: The Power of the Exponent. arXiv preprint arXiv:2208.09225, 2022.

[54] P. Lara-Benítez, M. Carranza-García, and J. C. Riquelme. An experimental review on deep learning architectures for time series forecasting. International Journal of Neural Systems, 31(03), 2021.

[55] I. Lauriola, A. Lavelli, and F. Aiolli. An introduction to deep learning in natural language processing: Models, techniques, and tools. Neurocomputing, 470:443–456, 2022.

[56] D. Lee and J.-W. Choi. DeFTAN-II: Efficient multichannel speech enhancement with subgroup processing. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 2024.

[57] J. Lee, J. Bae, B. Kim, S. J. Kwon, and D. Lee. To fp8 and back again: Quantifying the effects of reducing precision on llm training stability. CoRR, 2024.

[58] J. Lee, S. Markovich-Golan, D. Ohayon, Y. Hanani, G. Park, B. Kim, A. Karnieli, U. Livne, H. Shen, T. Huang, et al. Faster inference of llms using fp8 on the intel gaudi. arXiv preprint arXiv:2503.09975, 2025.

[59] B. Lim and S. Zohren. Time-series forecasting with deep learning: a survey. Philosophical Transactions of the Royal Society A, 379(2194), 2021.

[60] J. Lin, A. J. d. L. van Wijngaarden, K.-C. Wang, and M. C. Smith. Speech enhancement using multi-stage self-attentive temporal convolutional networks. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29:3440–3450, 2021.

[61] L. Liu, L. Tian, Z. Kang, and T. Wan. Spacecraft anomaly detection with attention temporal convolution networks. Neural Computing and Applications, 35(13):9753–9761, 2023.

[62] P. Liu, A. Hemani, K. Paul, C. Weis, M. Jung, and N. Wehn. 3D-stacked many-core architecture for biological sequence analysis problems. International journal of parallel programming, 45:1420–1460, 2017.

[63] W. Liu and F. Lombardi. Approximate Computing. Springer, 2022.

[64] J. L. Lobo, J. Del Ser, and F. Herrera. LUNAR: Cellular automata for drifting data streams. Information Sciences, 543:467–487, 2021.

[65] L. Lu, N. Kanda, J. Li, and Y. Gong. Streaming end-to-end multi-talker speech recognition. IEEE Signal Processing Letters, 28:803–807, 2021.

[66] Y. Luo, Z. Chen, and T. Yoshioka. Dual-path rnn: Efficient long sequence modeling for time-domain single-channel speech separation. In 2020 IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP), pages 46–50, 2020.

[67] Y. Luo and N. Mesgarani. Conv-TasNet: Surpassing Ideal Time–Frequency Magnitude Masking for Speech Separation. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 27(8):1256–1266, Aug. 2019.

[68] Q. Ma, Z. Liu, Z. Zheng, Z. Huang, S. Zhu, Z. Yu, and J. T. Kwok. A survey on time-series pre-trained models. IEEE Transactions on Knowledge and Data Engineering, 2024.

[69] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini. A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing. In 2018 IEEE Int. Symposium on Circuits and Systems (ISCAS), pages 1–5, 2018.

[70] S. Mach, F. Schuiki, F. Zaruba, and L. Benini. FPnew: An Open-Source Multiformat Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 29(4):774–787, 2021.

[71] 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), pages 1105–1110. IEEE, 2018.

[72] Y. Mao, X. Yu, K. Huang, Y.-J. Angela Zhang, and J. Zhang. Green edge ai: A contemporary survey. Proceedings of the IEEE, 112(7):880–911, 2024.

[73] B. Mazzoni, G. Tagliavini, L. Benini, and S. Benatti. An optimized heart rate detection system based on low-power microcontroller platforms for biosignal processing. In International Conference on System-Integrated Intelligence, pages 160–170. Springer, 2022.

[74] T. Meuser, L. Lovén, M. Bhuyan, S. G. Patil, S. Dustdar, A. Aral, S. Bayhan, C. Becker, E. d. Lara, A. Y. Ding, J. Edinger, J. Gross, N. Mohan, A. D. Pimentel, E. Rivière, H. Schulzrinne, P. Simoens, G. Solmaz, and M. Welzl. Revisiting edge ai: Opportunities and challenges. IEEE Internet Computing, 28(4):49–59, 2024.

[75] D. Michelsanti, Z.-H. Tan, S.-X. Zhang, Y. Xu, M. Yu, D. Yu, and J. Jensen. An overview of deep-learning-based audio-visual speech enhancement and separation. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29:1368–1396, 2021.

[76] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training, 2018.

[77] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training, 2018.

[78] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu. FP8 Formats for Deep Learning, 2022.

[79] P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu, and H. Wu. FP8 Formats for Deep Learning, 2022.

[80] U. I. Minhas, J. Lee, L. Mukhanov, G. Karakonstantis, H. Vandierendonck, and R. Woods. Increased leverage of transprecision computing for machine vision applications at the edge. Journal of Signal Processing Systems, 94(10):1101–1118, 2022.

[81] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. IEEE Communications Surveys & Tutorials, 20(4):2923–2960, 2018.

[82] F. Montagna, S. Mach, S. Benatti, A. Garofalo, G. Ottavi, L. Benini, D. Rossi, and G. Tagliavini. A Low-Power Transprecision Floating-Point Cluster for Efficient Near-Sensor Data Analytics. IEEE Transactions on Parallel and Distributed Systems, 33(5):1038–1053, 2021.

[83] mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 1.3.0), 2023. http://mpmath.org/.

[84] M. Z. Naser and A. H. Alavi. Error Metrics and Performance Fitness Indicators for Artificial Intelligence and Machine Learning in Engineering and Sciences. Architecture, Structures and Construction, nov 2021.

[85] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. P. Jouppi, and D. A. Patterson. The design process for google's training chips: Tpuv2 and tpuv3. IEEE Micro, 41(2):56–63, 2021.

[86] B. Noune, P. Jones, D. Justus, D. Masters, and C. Luschi. 8-bit Numerical Formats for Deep Neural Networks, 2022.

[87] NVIDIA. Transformer engine. https://github.com/NVIDIA/TransformerEngine, 2022.

[88] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In Proc. of the 50th Annual IEEE/ACM Int. Symposium on Microarchitecture, pages 41–54, 2017.

[89] R. Pascanu, T. Mikolov, and Y. Bengio. On the difficulty of training Recurrent Neural Networks, 2013.

[90] V. Pătrăucean, X. O. He, J. Heyward, C. Zhang, M. S. Sajjadi, G.-C. Muraru, A. Zholus, M. Karami, R. Goroshin, Y. Chen, et al. TRecViT: A Recurrent Video Transformer. arXiv preprint arXiv:2412.14294, 2024.

[91] H. Peng, K. Wu, Y. Wei, G. Zhao, Y. Yang, Z. Liu, Y. Xiong, Z. Yang, B. Ni, J. Hu, et al. Fp8-lm: Training fp8 large language models. arXiv preprint arXiv:2310.18313, 2023.

[92] S. P. Perez, Y. Zhang, J. Briggs, C. Blake, J. Levy-Kramer, P. Balanca, C. Luschi, S. Barlow, and A. W. Fitzgibbon. Training and inference of large language models using 8-bit floating point, 2023.

[93] A. Petitet, R. C. Whaley, J. J. Dongarra, and A. Cleary. Hpl—a portable implementation of the high-performance LINPACK benchmark for distributed-memory computers. [Online]. Available: http://www.netlib.org/benchmark/hpl/.

[94] A. Pullini, D. Rossi, I. Loi, G. Tagliavini, and L. Benini. Mr. wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. IEEE Journal of Solid-State Circuits, 54(7):1970–1981, 2019.

[95] M. Risso, A. Burrello, F. Conti, L. Lamberti, Y. Chen, L. Benini, E. Macii, M. Poncino, and D. Jahier Pagliari. Lightweight Neural Architecture Search for Temporal Convolutional Networks at the Edge. IEEE Transactions on Computers, 72(3):744–758, 2023.

[96] M. Risso, A. Burrello, D. J. Pagliari, F. Conti, L. Lamberti, E. Macii, L. Benini, and M. Poncino. Pruning In Time (PIT): A Lightweight Network Architecture Optimizer for Temporal Convolutional Networks. In 2021 58th ACM/IEEE Design Automation Conference (DAC), page 1015–1020. IEEE, 2021.

[97] A. Rix, J. Beerends, M. Hollier, and A. Hekstra. Perceptual evaluation of speech quality (PESQ)-a new method for speech quality assessment of telephone networks and codecs. In 2001 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, volume 2, pages 749–752, 2001.

[98] D. Rossi, F. Conti, M. Eggiman, A. Di Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, et al. Vega: A ten-core SoC for IoT endnodes with DNN acceleration and cognitive wake-up from MRAM-based state-retentive sleep mode. IEEE Journal of Solid-State Circuits, 57(1):127–139, 2021.

[99] D. Rossi, F. Conti, M. Eggiman, A. D. Mauro, G. Tagliavini, S. Mach, M. Guermandi, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini. Vega: A Ten-Core SoC for IoT Endnodes With DNN Acceleration and Cognitive Wake-Up From MRAM-Based State-Retentive Sleep Mode. IEEE Journal of Solid-State Circuits, 57(1):127–139, 2022.

[100] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. Pulp: A parallel ultra low power platform for next generation iot applications. In 2015 IEEE Hot Chips 27 Symposium (HCS), pages 1–39. IEEE Computer Society, 2015.

[101] B. Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. C. Koppaka, X. Song, S. Som, K. Das, S. Tiwary, S. Reinhardt, S. Lanka, E. Chung, and D. Burger. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. In Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[102] B. D. Rouhani, R. Zhao, A. More, M. Hall, A. Khodamoradi, S. Deng, D. Choudhary, M. Cornea, E. Dellinger, K. Denolf, S. Dusan, V. Elango, M. Golub, A. Heinecke, P. James-Roxby, D. Jani, G. Kolhe, M. Langhammer, A. Li, L. Melnick, M. Mesmakhosroshahi, A. Rodriguez, M. Schulte, R. Shafipour, L. Shao, M. Siu, P. Dubey, P. Micikevicius, M. Naumov, C. Verrilli, R. Wittig, D. Burger, and E. Chung. Microscaling data formats for deep learning, 2023.

[103] M. Rusci, M. Fariselli, M. Croome, F. Paci, and E. Flamand. Accelerating RNN-based Speech Enhancement on a Multi-Core MCU with Mixed FP16-INT8 Post-Training Quantization, 2022.

[104] O. Rybakov, N. Kononenko, N. A. Subrahmanya, M. Visontai, and S. Laurenzo. Streaming keyword spotting on mobile devices. ArXiv, abs/2005.06720, 2020.

[105] N. Saleem, T. S. Gunawan, S. Dhahbi, and S. Bourouis. Time domain speech enhancement with CNN and time-attention transformer. Digital Signal Processing, 147:104408, 2024.

[106] N. Saleem, T. S. Gunawan, M. Kartiwi, B. S. Nugroho, and I. Wijayanto. NSE-CATNet: deep neural speech enhancement using convolutional attention transformer network. IEEE Access, 2023.

[107] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. SIGPLAN Not., 46(6):164–174, jun 2011.

[108] N. N. Schraudolph. A fast, compact approximation of the exponential function. Neural Computation, 11(4):853–862, 1999.

[109] M. A. Shami and A. Hemani. Partially reconfigurable interconnection network for dynamically reprogrammable resource array. In 2009 IEEE 8th Int. Conference on ASIC, pages 122–125, 2009.

[110] N. Shoghi, A. Bersatti, M. Qureshi, and H. Kim. Smaq: Smart quantization for dnn training by exploiting value clustering. IEEE Computer Architecture Letters, 20(2):126–129, 2021.

[111] R. Singh and S. S. Gill. Edge ai: A survey. Internet of Things and Cyber-Physical Systems, 3:71–92, 2023.

[112] A. Skillman and T. Edsö. A Technical Overview of Cortex-M55 and Ethos-U55: Arm's Most Capable Processors for Endpoint AI. In 2020 IEEE Hot Chips 32 Symposium (HCS), pages 1–20. IEEE, 2020.

[113] S. Sonning, C. Schüldt, H. Erdogan, and S. Wisdom. Performance Study of a Convolutional Time-Domain Audio Separation Network for Real-Time Speech Denoising. In 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 831–835, 2020.

[114] A. Sorna, X. Cheng, E. D'Azevedo, K. Won, and S. Tomov. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW), pages 3–7. IEEE, 2018.

[115] V. Springel, R. Pakmor, O. Zier, and M. Reinecke. Simulating cosmic structure formation with the <scp>gadget</scp>-4 code. Monthly Notices of the Royal Astronomical Society, 506(2):2871–2949, July 2021.

[116] D. Stathis, Y. Yang, S. Tewari, A. Hemani, K. Paul, M. Grabherr, and R. Ahmad. Approximate Computing Applied to Bacterial Genome Identification using Self-Organizing Maps. In 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 560–567, 2019.

[117] T. Stuart, J. Hanna, and P. Gutruf. Wearable devices for continuous monitoring of biosignals: Challenges and opportunities. APL bioengineering, 6(2), 2022.

[118] C. Surianarayanan, J. J. Lawrence, P. R. Chelliah, E. Prakash, and C. Hewage. A survey on optimization techniques for edge artificial intelligence (ai). Sensors, 23(3), 2023.

[119] R. Szeliski. Computer vision: algorithms and applications. Springer Science & Business Media, 2010.

[120] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini. A Transprecision Floating-Point Platform for Ultra-Low Power Computing, 2018.

[121] 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), pages 1051–1056, 2018.

[122] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini. Design and evaluation of SmallFloat SIMD extensions to the RISC-V ISA. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 654–657. IEEE, 2019.

[123] G. Tagliavini, A. Marongiu, and L. Benini. Flexfloat: A software library for transprecision computing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(1):145–156, 2018.

[124] M. A. Tajammul, M. A. Shami, A. Hemani, and S. Moorthi. NoC Based Distributed Partitionable Memory System for a Coarse Grain Reconfigurable Architecture. In 2011 24th Int. Conference on VLSI Design, pages 232–237, 2011.

[125] K. Tan and D. Wang. Towards Model Compression for Deep Learning Based Speech Enhancement. IEEE/ACM Transactions on Audio, Speech, and Language Processing, 29:1785–1794, 2021.

[126] G. Thakur, H. Sohal, and S. Jain. Implementation of approximate multiplier using inexact compressors. In 2021 6th International Conference on Signal Processing, Computing and Control (ISPCC), pages 165–170, 2021.

[127] E. Tzinis, Z. Wang, and P. Smaragdis. Sudo rm-rf: Efficient networks for universal audio source separation. In 2020 IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP), pages 1–6. IEEE, 2020.

[128] C. Valentini-Botinhao et al. Noisy speech database for training speech enhancement algorithms and tts models. University of Edinburgh. School of Informatics. Centre for Speech Technology Research (CSTR), 2017.

[129] J.-M. Valin. A Hybrid DSP/Deep Learning Approach to Real-Time Full-Band Speech Enhancement, 2018.

[130] M. van Baalen, A. Kuzmin, S. S. Nair, Y. Ren, E. Mahurin, C. Patel, S. Subramanian, S. Lee, M. Nagel, J. Soriaga, and T. Blankevoort. FP8 versus INT8 for efficient deep learning inference, 2023.

[131] F. Váňa, P. Düben, S. Lang, T. Palmer, M. Leutbecher, D. Salmond, and G. Carver. Single precision in weather forecasting models: An evaluation with the ifs. Monthly Weather Review, 145(2):495–502, 2017.

[132] A. Vaswani. Attention is all you need. Advances in Neural Information Processing Systems, 2017.

[133] S. Wang and P. Kanwar. Bfloat16: The secret to high performance on cloud tpus. Google Cloud Blog, Aug. 2019. Accessed: 2025-10-15.

[134] P. Warden and D. Situnayake. TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. O'Reilly Media, 2019.

[135] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate computing: A survey. IEEE Design & Test, 33:8–22, 2016.

[136] Y. Yang, D. Stathis, P. Sharma, K. Paul, A. Hemani, M. G. Grabherr, and R. Ahmad. RiBoSOM: rapid bacterial genome identification using self-organizing map implemented on the synchoros SiLago platform. Proc. of the 18th Int. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2018.

[137] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. Axbench: A multiplatform benchmark suite for approximate computing. IEEE Design & Test, 34:60–68, 2017.

[138] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of Things for Smart Cities. IEEE Internet of Things Journal, 1(1):22–32, 2014.

[139] D. Zoni and A. Galimberti. Cost-effective fixed-point hardware support for risc-v embedded systems. Journal of Systems Architecture, 126:102476, 2022.