IoT and Edge Computing using virtualized low-resource integer Machine Learning with support for CNN, ANN, and Decision Trees

Towards Learning Technical Systems

Stefan Bosse

sbosse@uni-bremen.de

University of Bremen, Dept. Mathematics & Computer Science, Bremen, Germany

University of Siegen, Dept. Mechanical Engineering, Siegen, Germany

Overview

Tiny Machine Learning is a new approach that is being used for data-driven prediction, classification, and regression on microcontrollers using local sensor data.

Overview

Tiny Machine Learning is a new approach that is being used for data-driven prediction, classification, and regression on microcontrollers using local sensor data.

But even simple sensor data acquisition, aggregation, and processing is a challenge in distributed sensor network environments, the IoT, mobile networks, and other distributed strongly **heterogeneous** networks.

Overview

Tiny Machine Learning is a new approach that is being used for data-driven prediction, classification, and regression on microcontrollers using local sensor data.

But even simple sensor data acquisition, aggregation, and processing is a challenge in distributed sensor network environments, the IoT, mobile networks, and other distributed strongly **heterogeneous** networks.

The goal is to process sensor data locally and derive compressed relevant information features (e.g., damages, attacks, ...) with final global feature fusion.

Overview

To overcome issues and limitations with software and ML deployment in strong heterogeneous computer networks, the real-time capable low-resource Virtual Machine REXAVM is introduced.

Overview

To overcome issues and limitations with software and ML deployment in strong heterogeneous computer networks, the real-time capable low-resource Virtual Machine REXAVM is introduced.

REXA VM provides Virtualization of basic ML operations and models including but limited to: Decision Trees, ANN, CNN

Overview

To overcome issues and limitations with software and ML deployment in strong heterogeneous computer networks, the real-time capable low-resource Virtual Machine REXAVM is introduced.

REXA VM provides Virtualization of basic ML operations and models including but limited to: Decision Trees, ANN, CNN

REXA VM and its ML operations can be deployed on low-resource microncontrollers like the STM32 ARM Cortex M-series starting with 20 kB of RAM and 32 kB ROM only!

Introduction

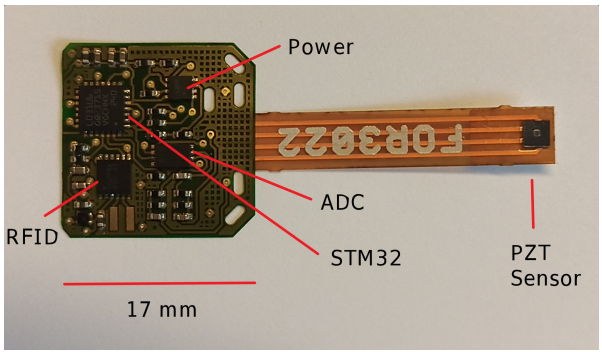


Fig. 1. Let's start here: A material-integrateable sensor node for damage diagnostics in Fibre-Metal Laminates using Guided Ultrasonic Waves (STM32 ARM Cortex M0, RFID, ADC) [IMSAS Bremen, B. Lüssem et al., 2023]

Host Platforms and Efficiency

Efficiency of data processing is always an important objective to optimize, especially for material-integrated sensor networks. The efficiency of data processing systems can be compared by the following normalized performance factor ε:

$$\epsilon = \frac{C \cdot M}{A \cdot P}$$

C: Data processing system's computational power in instructions per second (MIPS)

M: Memory capacity (RAM/ROM) in k Bytes

A: Entire chip area in mm^2

P: Electrical power consumption in mW.

Host Platforms and Efficiency

Device	Chip Area	Clock/MIPS	Power	RAM/ROM	ε
Atmel Tiny 20	2.1 mm^2 (1.55x1.4x0.53 mm)	12 MHz	4 mW	0.1 kB/2 kB	3
ARM Cortex M0 (Smart Dust 2002)	0.1 mm^2	740 kHz	70 mW	4 kB/4 kB	0.84
FreeSclae KL03 (ARM Cortex M0+)	4 mm^2	48 MHz	3 mW	2 kB/40kB	168
STM32 F103VC M3	~10 mm^2	72 MHZ	200 mW	48 kB/256 kB	11
STM32 F103C8 M3	~6 mm^2 (meas.)	48 MHZ	100 mW	20 kB/64 kB	6.7
STM32 L031G6U6 M0+	0.25 mm^2 (meas.)	16 MHZ	2 mW	8 kB/32 kB	1280
STM32 L073CZU6 M0+	~1 mm^2	16/32 MHZ	5/12 mW	20 kB/192 kB	678/565
Xilinx Spartan 3-500E	9.6 mm^2 (meas.)	50 MHz	100 mW	45 kB	2.34
Xilinx Spartan 7-S25	~50 mm^2	100 MHz	100 mW	202 kB	4

The Concepts

1. VM with integrated compiler
2. Programs (and ANN models, too) are always delivered in textual format
3. On-the-fly compilation to linear Bytecode (< 600 lines of C code!)
4. No dynamic memory management except by stack operations
5. KISS (< 3000 lines of C code); highly configurable (custom ISA)
6. VM can be directly embedded in IO loops (microcontrollers) cooperating with other tasks

VM Architecture

Memory Model and Instruction Processing

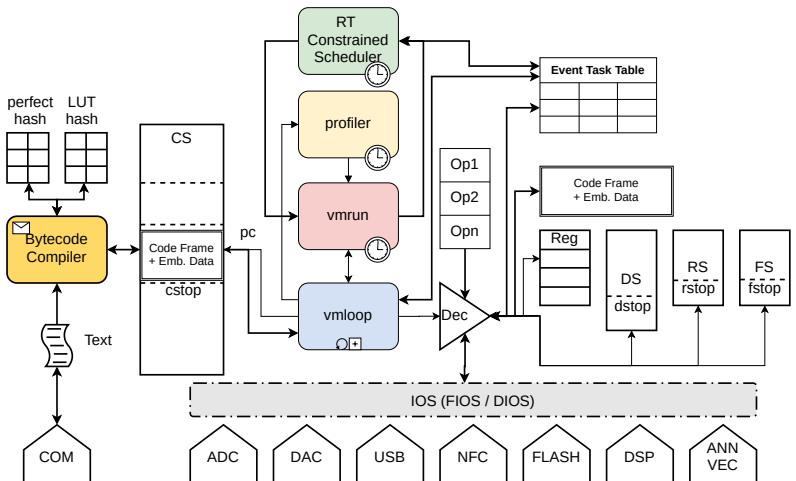


Fig. 2. Multi-stack Computer with mixed-mode code segment (no heap memory), integrated JIT compiler, and Bytecode processor (vmloop)

Instruction Set Architecture

- Most ops are zero-operand instructions (single world) operating directly on the stack(s) or the program counter
- With some exceptions the ISA can be freely defined (via code snippets and macro definitions, discussed in the SDK section)
- Zero-operand operations consume one Byte (see next slide)
- Most instructions have constant and equal execution times (real-time; run-time prediction possible)

But the widely used and well known FORTH programming language will be used commonly (or any sub-set; there is no real standard)

FORTH

- Reverse Polish Notation (stack language)
- "Write once and forget (read never)" issue
- But keeps compiler simple (low resources and compilation times)

```
var x
10 20 + x !
x @ . cr
: vecmean
  0
  100 0 do
    data i cell+ @ +
  loop
;
vecmean . cr
```

Pocket GUW Laboratory

- Example of an application using the call-gate interface: A digital oscilloscope equipped with the REXA VM

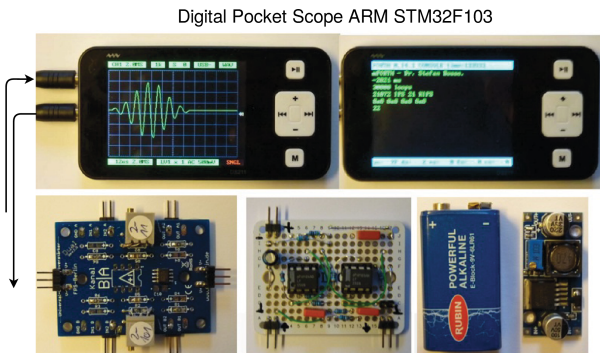


Fig. 3. The pocket GUW laboratory only using low-budget and low-quality devices for GUW-based damage detection in Fibre composite materials. The DSO implements REXA-VM and communicates via an USB virtCOM port with an external computer. https://arxiv.org/abs/2302.09002v1

VM Tiny ML

ANN and CNN computations require efficient and generic vector operations crucial to implement ML on microcontrollers. The REXA VM provides a core set of vector operations that can be used for the computation of ANN and CNN models.

VM Tiny ML

ANN and CNN computations require efficient and generic vector operations crucial to implement ML on microcontrollers. The REXA VM provides a core set of vector operations that can be used for the computation of ANN and CNN models.

Training using classical error back-propagation is currently not supported due to the requirement of storing a suitable training and test data set on the device.

Vector Operations

- Only integer arithmetic is supported (adressing low-resource and low-power microcontrollers)

An ANN (and CNN) consists of two parts:

1. The data, i.e., for parameter, input, and output variables;
2. The structure and functions processing the data.

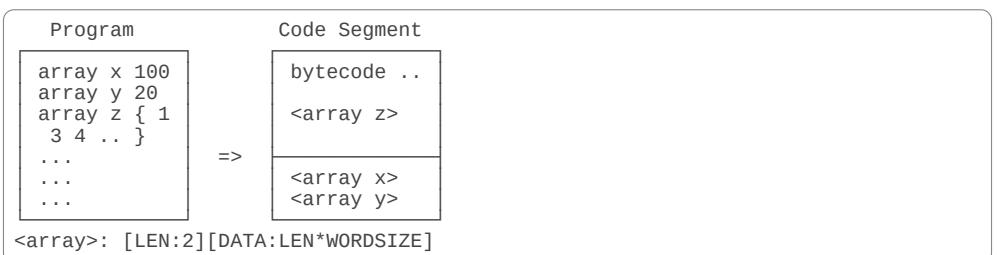
The ANN can be functionally decomposed into the following vector and matrix operations assuming integer approximation:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^p \approx \mathbb{I}^n \rightarrow \mathbb{I}^p, f = g \circ f_{l-1} \circ f_{l-2} \circ .. \circ f_1, f_i(\vec{x}) = a\left(\hat{w}_i \vec{x} + \vec{b}_i\right)$$

$$g(\vec{z}) = \begin{cases} z & \text{regression} \\ \frac{1}{1+e^{-z}} & \text{binary classification} \\ \frac{e^{z_j}}{\sum_k \left(e^{-z_k}\right)} & \text{multi-classification} \end{cases}$$

Vector Operations

- ANN models can be decomposed in chained vector operations!
- Vectors are initialised arrays (model parameters) or initialised arrays (input, intermediate, and output data)
- Vector (array) data is embedded in the Code Segment (no heap!)



Def. 1. Initialized arrays embedded in-place in code frames and non-initialized arrays stored at the end of the compiled code frame

Vector Operations

- For ANN and CNN models, a set of scaled vector (array) operations is provided (commonly W=16 Bits signed integer).
- Most vector operations are using $2W$ arithmetics internally (e.g., 32 Bits) with final down (or up) scaling of results
- Scaling parameters must be computed by a model analyzer

Operation	Description
`array`	Create an initialised or unintialised array (vector)
`vecscale`	Scale a vector (negative scale value: division, positive: multiplication)
`vecadd vecmul`	Elementwise vector addition and multiplication
`vecfold`	Folding operation (ANN FC layer for multiple neurons)
`vecconv`	Multi-purpose convolution and pooling operation (CNN)
`vecmap`	Elementwise application of a function (e.g., relu or sigmoid), used for ANNs and CNNs
`vecreduce`	Vector reduction (scalar output), e.g., minimum or maximum search, sum, product

Activation Functions

There are different transfer (activation) that are used in ANN and CNN modells, mosr prominent examples are:

- Linear function (*linear*) without x- and y-limits
- Logistic or sigmoid function (*sigmoid*) with y-limit=[-1,1]
- Tangents hyperbolic function (*tanh*) with y-limit=[-1,1]
- Rectifying linear unit (*relu*) with one-side open y-limit=[0,∞)

The *linear* and *relu* functions can be directly implemented with integer arithmetic without loss of accuracy (except due to integer discretizing).

Activation Functions

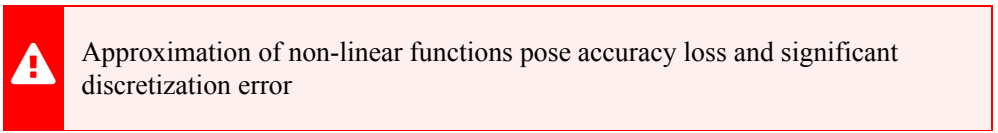
There are different transfer (activation) that are used in ANN and CNN modells, mosr prominent examples are:

- Linear function (*linear*) without x- and y-limits
- Logistic or sigmoid function (*sigmoid*) with y-limit=[-1,1]
- Tangents hyperbolic function (*tanh*) with y-limit=[-1,1]
- Rectifying linear unit (*relu*) with one-side open y-limit=[0,∞)

The *linear* and *relu* functions can be directly implemented with integer arithmetic without loss of accuracy (except due to integer discretizing).

The highly non-linear *sigmoid* and *tanh* functions require an appropriate approximation by using a hybrid approach using a (compacted) look-up table (LUT) and interpolation.

Activation Functions



Approximation of non-linear functions pose accuracy loss and significant discretization error

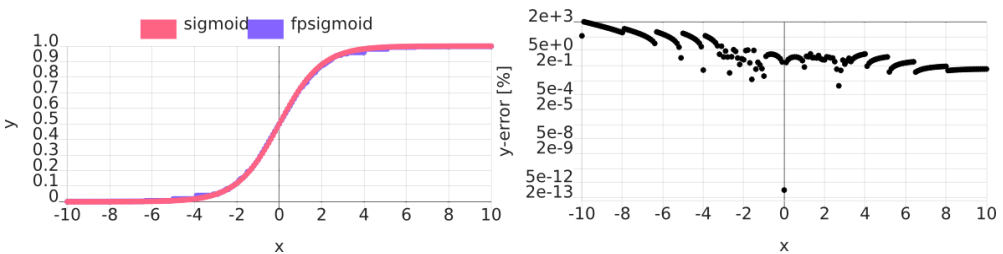


Fig. 4. Relative discretization error of scaled integer LUT-interpolated approximation of the *sigmoid* function

ANN Template

Model Data

```
( Layers: 14,8,2 #neurons:24 )
array input 14
( Layer I )
array wghtsI { 1 1 ... 1 1 }
array biasI  { 1 1 ... 1 1 }
array scaleI { 1 1 ... 1 1 }
array actI 14
( Layer H1 )
array wghtsH1 { 1 1 ... 1 1 }
array scaleH1 { 1 1 ... 1 1 }
array actH1 8
( Layer O )
array wghtsO { 1 1 ... 1 1 }
array biasO  { 1 1 }
array scaleO { 1 1 }
array output 2
```

Model Computation

```
( Input data is stored in input )
( Output data is stored in output )
: forward
  ( Layer I )
  input wghtsI actI scaleI vecmul
  actI biasI actI 0 vecadd
  actI actI $ sigmoid 0 vecmap
  ( Layer H1 )
  actI wghtsH1 actH1 scaleH1 vecfold
  actH1 biasH1 actH1 0 vecadd
  actH1 actH1 $ sigmoid 0 vecmap
  ( Layer O )
  actH1 wghtsO output scaleO vecfold
  output biasO output 0 vecadd
  output output $ sigmoid 0 vecmap
;
```

CNN Template

Model Data

```
( Layers: conv,pool,fc )
array input 250
( Layer 1 conv )
array cK0L1 { 1 1 ... 1 1 }
array cK1L1 { 1 1 ... 1 1 }
array cK2L1 { 1 1 ... 1 1 }
array cSL1  { 1 1 1 }
array cOL1 104
( Layer 2 pool )
array pO0L2 12
array pO1L2 12
array pO2L2 12
( Layer 3 fc )
array fW0L3P0 { 1 1 ... 1 1 }
array fW0L3P1 { 1 1 ... 1 1 }
array fW0L3P2 { 1 1 ... 1 1 }
array fW1L3P0 { 1 1 ... 1 1 }
array fW1L3P1 { 1 1 ... 1 1 }
array fW1L3P2 { 1 1 ... 1 1 }
array fAL3 12
array fBL3 { 1 1 }
array fSL3 { 1 1 }
array fOL3 2
array output 2
( Input data is stored in input )
( Output data is stored in output )
```

Model Computation

```
: forward
  ( Layer 1 conv )
  ( merged with Layer 2 pool )
  input cK0L1 cOL1 cSL1 0 cell+ @ 50 3 2 2 vecconv
  cOL1 cOL1 $ relu 0 vecmap
  cOL1 256 3 + pO0L2 0 26 -3 2 0 vecconv
  input cK1L1 cOL1 cSL1 1 cell+ @ 50 3 2 2 vecconv
  cOL1 cOL1 $ relu 0 vecmap
  cOL1 256 3 + pO1L2 0 26 -3 2 0 vecconv
  input cK2L1 cOL1 cSL1 2 cell+ @ 50 3 2 2 vecconv
  cOL1 cOL1 $ relu 0 vecmap
  cOL1 256 3 + pO2L2 0 26 -3 2 0 vecconv
  ( Layer 3 fc )
  pO0L2 fW0L3P0 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO1L2 fW0L3P1 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO2L2 fW0L3P2 fAL3 0 vecmul
  fAL3 0 12 8  vecreduce
  2+ 2+ fSL3 0 cell+ @ 2ext 2/ 2red sigmoid
  fOL3 0 cell+ !
  pO0L2 fW1L3P0 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO1L2 fW1L3P1 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  pO2L2 fW1L3P2 fAL3 0 vecmul
  fAL3 0 12 8 vecreduce
  2+ 2+ fSL3 1 cell+ @ 2ext 2/ 2red sigmoid
  fOL3 1 cell+ !
;
```

Software Development Kit

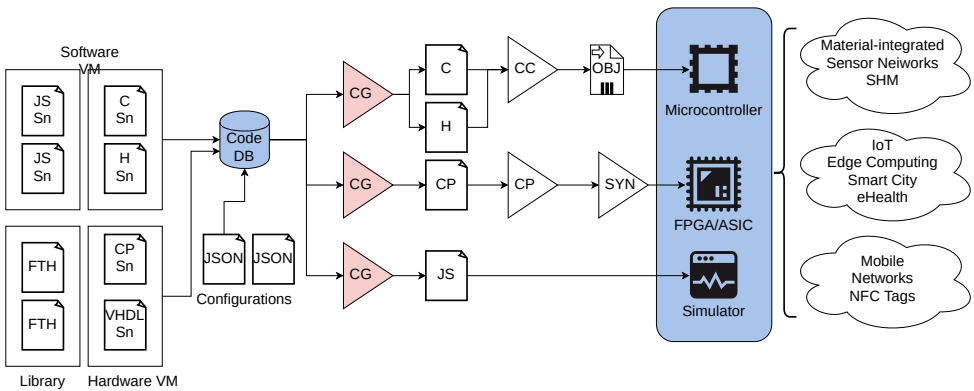


Fig. 5. Overview of the overall concept of REXA-VM development (C-SN: C source code snippet, H-SN: C Header snippet, JS: JavaScript, FTH: Forth VM code definitions, JSON: JavaScript Object Notation, CG: Code generator, CC: C Compiler, CP: ConPro HLS, SYN: RTL synthesis tool)

- The ISA is defined by a collection of code snippets and macro definitions
  - There are different implementations for different host platforms (or OS)
  - There are different implementations for software and hardware VMs
- All code and definitions are stored in a simple JSON data-base that can be accessed by various compiler programs

Git it here and try out: https://github.com/bsLab/rexavm/

Use Cases

Damage Detection with an ANN

- In this use-case, aggregated feature variables derived from time-dependent Ultrasonic signals (Guided Ultrasonic Waves, GUW) from multi-path measurements were used to predict a damage in a composite materials and to estimate its location.

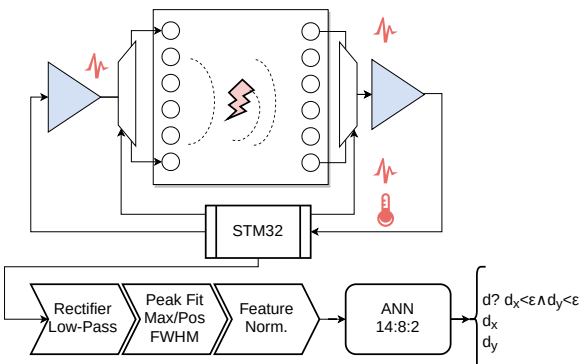


Fig. 6. Multi-path GUW measurement and data processing for damage detection (classification and location regression)

Damage Detection with an ANN

1. The feature variables were computed from the signal hull, mainly by analyzing the first main maximum (position and width).
2. The hull signal was computed using (1) the analytical signal via the Hilbert transform (using FFT) and (2) by applying a rectifier and low-pass filter. Only the second method can be implemented on the STM32 microcontroller.
3. Assuming six measuring paths and the two most significant feature variables normalized peak position and peak height, additionally using a measure temperature and the base frequency of the pitch signal, the feature vector consists of 14 variables in total.
4. This scaled feature vector is the input for a simple ANN (three layers, one hidden, typical layer structure [14,8,2], sigmoid activation functions).
5. The output of the ANN provided an estimation of the x- and y-coordinates of the damage location (or close to 0 if there is no damage detected). This is a hybrid classification and regression model. If only classification is required, one output neuron is sufficient.

Damage Detection with a CNN

- Similar to the previous use-case, single-path Ultrasonic time-dependent measuring signals are used to predict a damage in a composite material.
- A Convolutional Neural Network is used to predict a damage (binary classifier)
- Spectrogram-based method representing the frequency space of a time-dependent signal as an image

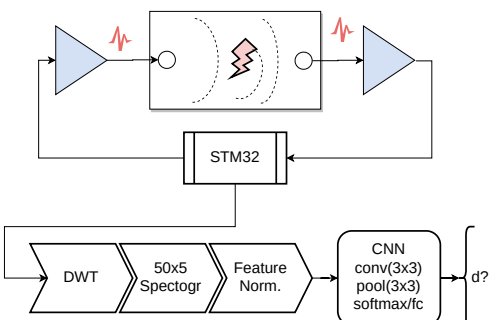


Fig. 7. Single-path GUW measurement and data processing using a CNN for damage detection

Damage Detection with a CNN

1. A discrete wavelet transform using high- and low-pass filters is used to decompose the sensor signal into wavelet coefficients (first 5 levels were chosen).
2. The output of the filters (detail and approximation) are decimated by a factor of two, retaining only the even samples, since each filter output contains half of the frequency content, but an equal amount of samples as the input signal.
3. With increasing level the number of data elements decreases by a factor 2. To provide the output of multiple levels in matrix form, the lower levels are shrink to the number of elements of the highest level (5).
4. The original signal window contained about 2000 samples, finally providing only 50 data points for the fifth DWT decomposition layer.
5. All DWT vectors are combined into a 50 × 5 elements matrix, treated as a two-dimensional spectogram image (Inout for CNN).
6. **Model is trained with FP arithmetics (classical error gradient back-propagation), then transformed to scaled integer model**.

Damage Detection with a CNN

Results

The original FP arithmetic model showed a classification error of 5%

Damage Detection with a CNN

Results

The original FP arithmetic model showed a classification error of 5%

The integer-scaled and transformed REXAVM model using the vector ISA showed a classification error of 5% (same data set)!

Damage Detection with a CNN

Results

The original FP arithmetic model showed a classification error of 5%

The integer-scaled and transformed REXAVM model using the vector ISA showed a classification error of 5% (same data set)!

But: Initial transformations only regarding overflow scaling showed underflow issues and high classifcation errors requiring a re-calibration of the scaling factors.

Performance

VM

1. Compilation (MCPS, Tokens)
2. Bytecode execution (MWPS, Bytecode Instructions)

Target	Configuration	MIPS	MCPS	Code/Heap
STM32 F103VC3, 72MHz, 256kB ROM, 48kB RAM	CS=1024, DS=256, RS=128, FS=64, Words=101	1.1 / 15k/MHz	0.1 / 1.4k/MHz	8/8 kB
STM32 F103VC3, 72MHz, 256kB ROM, 48kB RAM	CS=1024, DS=256, RS=128, FS=64, Words=64 (no double word arithmetic)	1.1	0.1	7/7 kB
STM32 F103VC3, 72MHz, 256kB ROM, 48kB RAM	CS=4096, DS=1024, RS=256, FS=128, Words=101	1.1	0.1	8/16 kB
STM32 L031, 16 MHz, 32 kB ROM, 8 kB RAM	CS=1024, DS=256, RS=32, FS=32, Words=101	0.24 / 15k/MHz	0.02	7.1/8 kB
i5-7300U, 3GHz 4 GB RAM	CS=16384, DS=4096, RS=1024, FS=256, Words=101	280 / 90k/MHz	27 / 9k/MHz	32/64 kB

VM

Highlights

- 1:70 → About 70 native machine instructions / VM instruction execution (ARM Cortex) or 1:15 (Intel x86)
- 1:700 → About 700 native machine instructions / Word compilation (ARM Cortex) or 1:100 (Intel x86)
- Only 13 nJ / VM instruction (ARM Cortex M0+)
- Only 130 nJ / Word compilation (ARM Cortex M0+)
- Computation times of medium sized ANNs is below 1 Second (ARM Cortex M0+, 16 MHz, typically in the Milliseconds range)
- Compilation times of medium sized programs is below 1 Second (typically in the Milliseconds range)
- Start-up time of VM is below 100 ms (typically in the Milliseconds range)

Vector Operations (ANN)

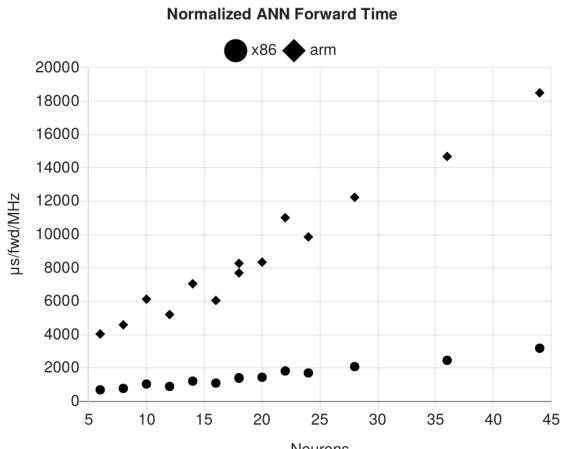


Fig. 8. Normalized computation times for ANNs of different size (with two, three, and four layers) and two different host platforms (Generic i5 x86 @2900 MHz and STM32F103C8 @72MHz) as a function of neurons: $\Theta(N)=N$

Vector Operations (ANN)

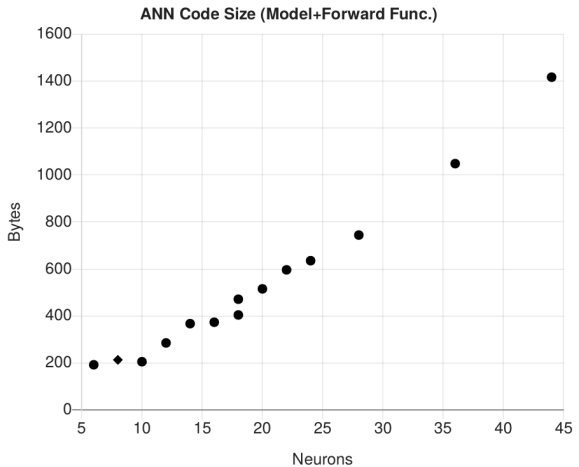


Fig. 9. Code size of ANN as a function of the number of neurons: $\Theta(N)=N$

Vector Operaiotns (CNN)

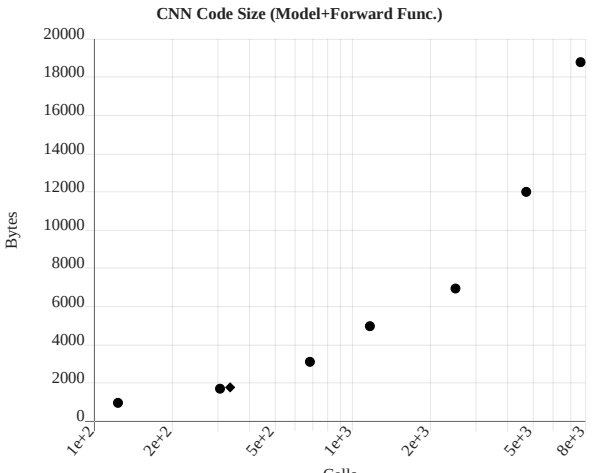


Fig. 10. Code size of CNN as a function of the number of cells: $\Theta(N)=N^2$

Summary

A stack-based virtual machine architecture for low-resource, tiny embedded systems was introduced and analyzed. The overhead, even on very low-resource systems, is low with respect to typical running times under energy constraints and tasks to be performed in real-time

Summary

A stack-based virtual machine architecture for low-resource, tiny embedded systems was introduced and analyzed. The overhead, even on very low-resource systems, is low with respect to typical running times under energy constraints and tasks to be performed in real-time

A major feature is the tight coupling of a Text-to-Bytecode compiler with the Bytecode interpreter, ensuring robustness, security, stability, and interoperability in strong heterogeneous environments.

Summary

A stack-based virtual machine architecture for low-resource, tiny embedded systems was introduced and analyzed. The overhead, even on very low-resource systems, is low with respect to typical running times under energy constraints and tasks to be performed in real-time

A major feature is the tight coupling of a Text-to-Bytecode compiler with the Bytecode interpreter, ensuring robustness, security, stability, and interoperability in strong heterogeneous environments.

ML classification and regression models can be computed using integer arithmetic and a set of vector operations with low computation times and memory requirements.

IoT and Edge Computing using virtualized low-resource integer Machine Learning with support for CNN, ANN, and Decision Trees

Towards Learning Technical Systems

Stefan Bosse

sbosse@uni-bremen.de

www.edu-9.de

University of Bremen, Dept. Mathematics & Computer Science, Bremen, Germany

University of Siegen, Dept. Mechanical Engineering, Siegen