Tiny Machine Learning Virtualization for IoT and Edge Computing using the REXA VM

Stefan Bosse
Dept. Mathematics and Computer Science
University of Bremen
Bremen, Germany
ORCID 0000-0002-8774-6141

Christoph Polle
Measurement Systems & Monitoring
Faserinstitut Bremen
Bremen, Germany
polle@Faserinstitut.de

Abstract— **Tiny Machine Learning is a new approach that is being used for data-driven prediction classification and regression on microcontrollers using local sensor data. The models are typically learned off-line and sent to the microcontroller for use as binary objects or frozen and converted static data. This approach is not universal or flexible. The REXA VM, which can virtualize embedded systems and sensor nodes and includes a general machine learning framework that supports arbitrary dynamic artificial neural network and decision tree models, is introduced in this study. The models are delivered as text files with highly compressed program code that are enclosed in code frames with embedded data (model parameters). The VM offers fundamental computations for ANN and DT models (Microservices). Using a decompiler, models can be updated (retrained) and sent to other nodes (mobile models). It can be demonstrated that virtualization using a bytecode machine and just-in-time compiler is still appropriate and effective for extremely low-resource processors.**

Keywords— Virtualization, Virtual Machines, Tiny ML, Sensor Networks, Embedded Systems, Microservices

I. Introduction

Advanced and reliable data processing architectures are needed to handle a large rise in device density and sensor deployment towards smart and self-*; sensors, addressing ubiquitous computing, edge computing, and distributed sensor networks. Tiny Machine Learning (ML) is an emerging and challenging field [1]. ML models are commonly computed using high precision floating point arithmetic. Tiny embedded systems provide only integer arithmetic (8-32 bits), requiring either model transformation and freezing [2] or direct training using integer arithmetic [3], ideally on the target device itself [4]. Ultra low-power devices introduce additional constraints on the computation of Deep learning (DL) models [5], and hardware designs gains attraction [6], which are addressed in this work, too.

This paper introduces and analyzes a real-time capable and extensible application-specific stack Virtual Machine (REXA-VM) with some unique and special features that can be implemented in tiny embedded systems with a microcontroller and as little as 8 KB data RAM and 16 kB code ROM. Virtualization in combination with Machine Learning (ML) is a necessity for unified sensor and data processing in large-scale and heterogeneous networks [7]. One major feature is a scriptable Tiny ML interface and signal analysis numeric using 16 bits scaled arithmetic. In contrast to common integer-based ML models using 8 bit scaled arithmetic [3], this VM supports 16 and 32 bit operations natively, preventing common arithmetic overflow and underflow issues. The real-time capability of REXA VM is important to ensure operational and event-reaction stability during time-consuming ML computations. The VM consists of a bytecode compiler and interpreter. The input is always in text format using a simple stack programming language similar to Forth, suitable for hardware implementations in embedded systems [8]. Using an ASCII text code and data format is a key feature required for the deployment of the REXA VM in highly heterogeneous environments including different host platform byte orders. One of the benefits of a virtualization layer is the freedom of implementation technologies. Virtualization and their limitations in embedded systems are discussed extensively in [9]. Therefore, an alternative implementation of the REXA-VM in FPGAs (or ASICs) with an RTL architecture can be provided, too. The **novelties** of this work are:

1. A unified and customizable software and hardware architecture of the scriptable and real-time capable stack-based VM (HW/SW co-design at the architecture level), supporting Tiny ML for highly compact Artificial Neural Networks (ANN) and decision tree (DT) computations;
2. Highly customizable machine Instruction Set Architecture (ISA) supporting 16 and 32 bit arithmetic;
3. High-speed just-in-time text-to-bytecode compiler (always bound to VM, code is always exchanged in text format) and optional bytecode-to-text decompiler supporting mobile code (and mobile ML models);
4. Data is embedded in code, not needing any heap and dynamic memory management except for code frames, e.g., embedding and storing an ANN and its parameter inside program code.

Non-continuous energy supply, e.g., supplied to the sensor node from outside sources using RFID/NFC, is another challenge introducing hard power constraints and a suitable degree of real-time capable data processing under energy and time constrains. The real-time capability of the VM (not discussed in this paper) is a key feature for running computational intensive tasks without compromising IO event handling (i.e., the reactivity of the device). It is assumed that a REXA VM node is accessed remotely via wireless communication. Program text code can be directly send to the VM for compiling and processing.

The following sections introduce the relevant features and architectures of the REXA VM, with a focus on ML. An extended use-case demonstrates finally the suitability of the presented virtualization and programming interface for on-device damage prediction. An extended pre-print version can be found in [14].

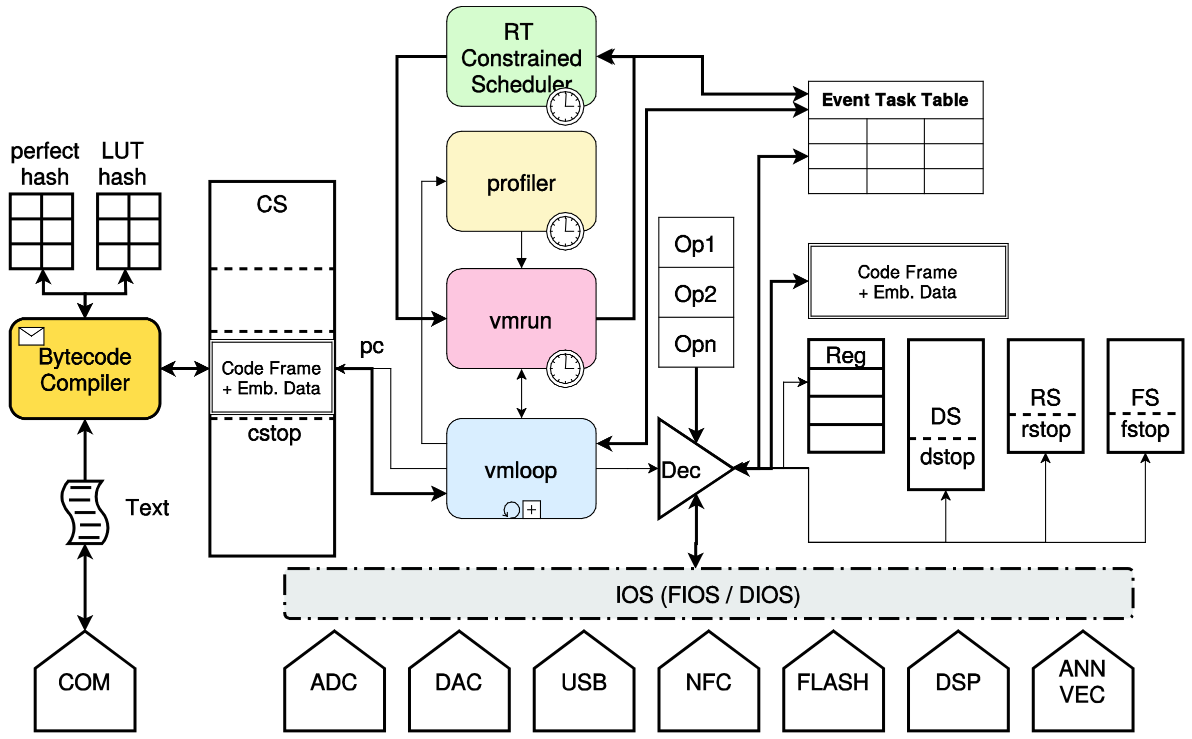


Fig. 1. Basic REXA-VM architecture with integrated JIT compiler, stacks, and bytecode processor

II. Virtual Machine Architecture

The REXA VM is a highly configurable high-level stack processor that executes bytecode generated from textual program code, ensuring high portability, flexibility, and robust code processing in heterogeneous environments (including the deployment of different versions of the REXA VM). Basically, the entire ISA can be customized, creating a balance between generality and application-specific optimizations. Most instructions are zero operand operations accessing data entirely on the VM stacks (in the program code model, they are post-fix instructions because data was stored by previous operations).

The following Fig. 1 shows the principle architecture of the REXA VM and its just-in-time (JIT) compiler. The architecture details depend on the configuration (single- or multi-tasking, number of stacks, and customized extensions and accelerators). The principle architecture is equal for software and hardware implementations. The native compiler is available in software and in hardware, but can be replaced by a lightweight pure Forth-based implementation. Profiling is an optional feature used for predictive real-time scheduling, as well as the energy-aware real-time scheduler.

The code segment (CS) is the central storage for source code, bytecode, and embedded data. The CS is partitioned into dynamically sized code frames, commonly assigned to a task (depending on the scheduling model). The scheduler controls and monitors the bytecode loop (*vmloop*). Code operations can suspend task execution by waiting for events, handled by an event table. The Input-Output System (IOS, similar to the widely used Foreign Function Interface, extends the code and data space of the VM). The VM architecture is optimized for resource sharing, e.g., using an ADC sample buffer for computations from VM programming level, too.

The main user program memory is the code segment of the VM is the code segment, (CS). A code segment is organized in byte cells and has a static fixed size. The new program code allocates a part of the CS, called a code frame (CF). A code frame can contain top-level operations, word definitions that can be added to the VM dictionary, and data variable allocations embedded in the code frame (there is no heap memory), either directly between Forth instructions or at the end of the code frame program (e.g., non-initialized array data). After the code frame program processing terminates (called the *end* operation), the code frame with all of its data is removed. Alternatively, the code frame can be kept alive after termination, and exported code word references can be used from later code frames. This feature enables incremental and partitioned program execution.

In single-tasking mode, the code segment is commonly incremental and persistent. That means, program code is added to the CS incrementally. Each new code fragment is compiled and immediately executed. The current program terminates with the *end* instruction, with or without persistence. Without persistence, the code is removed after termination. If the code fragment is persistent, a new code fragment is stored at the end of the previous one. Persistent code cannot be removed, only resetting the CS is possible. Only the current code fragment can be scheduled, without branching to other code regions. In multi-tasking mode, the code segment is partitioned into dynamically sized code frames. Code frames can be removed later, and code frames can be linked. Scheduling can branch to other code frames. Each code frame is associated with a task, except code frames that terminated with persistence.

A code frame merges code and data. The data (scalar and array variables) can only be accessed by code inside the code frame. Operations can be bound to named words, which can be exported in the global dictionary and accessed from other code frames (and tasks). If a function word is exported, the code frame is locked and is not removed if the code processing reaches the end (instruction).

Temporary (short lifetime) data is stored and manipulated directly on fixed-size stack memories:

1. The Data Stack (DS) holds most of the processing data and instruction operands;
2. The Return Stack (RS) for function calls (not accessible from the programming level for security reasons);
3. Optional a loop stack (FS) used for loop counters and secondary user data (can be merged with RS for memory efficiency).

All non-temporary data is either embedded in code frames or provided by the host application via the Input-Output System layer (IOS) API.

The stack cell width is always 16 bits (single word width). The REXA VM supports double word operations, too (as a configurable option). Double words are composed of two single data words (word order depends on the native byte order of the underlying processor). Double words can be directly read and written from and to stacks by the VM (single memory access). The access time of multiplexed single and double word access to the stacks in software by memory pointer casting is commonly identical (assuming 32-bit microprocessors). The hardware implementation can split the double word access into two memory cycles or use 32-bit memory for stacks, always providing one-cycle memory access. For performance reasons, the hardware implementation can implement stacks with block RAM components and single registers holding the first top values, enabling parallel computations on stack elements. The push and pop operations involved in most of the VM instruction code words modify stack pointers (*dstop*, *rstop*, *fstop*). For security reasons, the return stack (which holds code pointers on function calls) should not be accessed directly by program code.

Besides hardcore stacks implemented inside the VM, softcore stacks can be implemented on the programming level in data arrays (embedded in code frames). Push and pop operations are provided by the core instruction word set.

A. Compiler

The compiler translates the source code text into bytecode instructions. It is a just-in-time (JIT) compiler that can compile code incrementally and on demand. Since the ISA of stack processors consists mostly of zero-operand instructions, it supports fine-grained compilation at the token level. The source text can be directly stored in the code segment referenced by a code frame (or any other data buffer, alternatively). Most instruction words can be directly mapped to a consecutively numbered operation code. Therefore, the compiler translates the source code into bytecode directly in-place, i.e., by replacing the text with bytecode, saving additional target memory buffers. An instruction word consists of at least one character, and thus can always be replaced by the op-code (one byte). Although a literal value can consist of only one digit and the data of a single word value occupies two bytes, there is always a space or newline character after a literal value, providing the required data space. Extension of the current code frame at the end is always possible (as long as there is free space in the CS). One exception is a double word literal value requiring at least two characters and the suffix "l", followed by an obligatory separator character and the space, providing four bytes of data space in total.

Data is either stored on the stacks during run-time or is embedded in the code frame during translation. Scalar variables and initialized arrays can always be embedded in-place. Non-initialized arrays are appended at the end of the compiled code frame.

The compiler is part of the VM and is processed directly on the (low-resource) embedded system or in hardware. This ensures security checks and guarantees that a task assigned to a code frame can only access its private data. Finally, the textual interface ensures compatibility and interoperability among different VM versions (with different binary ISA, but a common sub-set). Alternatively, most parts of the compiler can be implemented in the high-level VM language itself. In this case, the VM only provides basic compiler operations like a tokenizer and a dictionary.

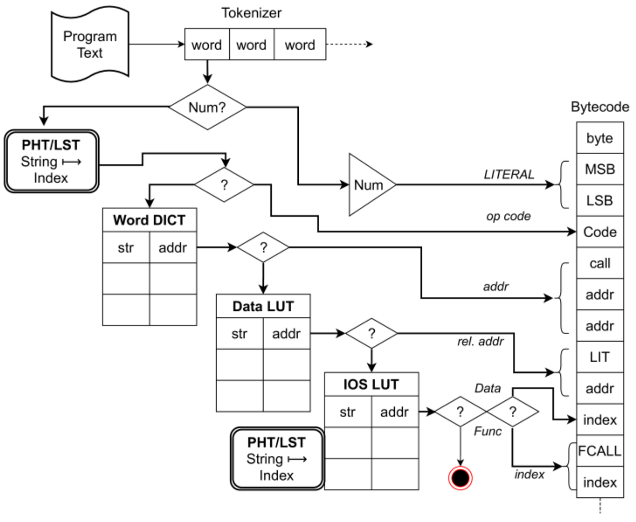


Fig. 2. The hierarchical compiler LUT architecture used for fast text-to-bytecode translation. After tokenization of the input text, direct op-code encoding and updates of tables are performed.

To enable fast compilation (with respect to low-resource microcontrollers), no traditional lexer and parser are used. Instead, the parsing process uses two different approaches:

1. A perfect hash table [10,11] (PHT) for core words (more than 100 core words) with a constant search time. Because the hash index is directly linked to the operational code, the hash table itself must be saved. A hash function, on the other hand, cannot detect words that do not match any of the core word set. Therefore, a string table is required containing all core words indexed by the same hash index, finally comparing a hash-predicted word from this table with the current word to be parsed.
2. A Linear Search Table (LST) with non-constant search time. The LST implements an iterative and sequential character search in a compacted linear array, as discussed below. The LST needs more (ROM) space but, on average, less machine instructions (basic operations) for search than the PHT.

There are additional dictionaries, e.g., the global instruction word dictionary. Dictionary words and local data variables are handled with simple hashing and small look-up tables (LUT) storing collisions by linear search. The hierarchical look-up table architecture for fast text-to-bytecode translation is shown in Fig. 2. The REXA VM bytecode consists of operational code and data. Most instructions are zero-operand post-fix operations, simplifying the bytecode format significantly (and the code decoder, especially addressing hardware implementations). Literals are divided into signed short and double words (data widths of 14 and 30 bits, respectively). A bytecode code frame can be decompiled to text again, supporting mobile code. Embedded initialized data, e.g., parameter arrays of an ANN, can be modified at run-time (update training), and the modified values are contained in the text code that can be send to another device.

B. *Bytecode Interpreter and Multitasking*

The VM architecture and code processing can be either single-tasked or multi-tasked (using the code frame mode). Although the VM can be replicated and share the same code segment and global objects, providing multi-threading (real multi-tasking) with parallel execution on multi-core processors or replicated RTL hardware, multi-tasking is a scheduling without concurrency, i.e., tasks are co-routines. The priority can be changed by the scheduler only if there are energy conflicts. There are event-based and computation-based tasks. Both differ in their run-time behaviour. Negative priorities indicate a short-running event-based IO task; positive priorities indicate a greedy computational task. Besides multi-tasking, which provides a scheduled programming and code execution model without parallelism, multiple VM instances can be easily composed into a parallel VM. Each parallel VM shares the same code interpreter (decoder) and code segment (and compiler, if implemented), but with individual stack segments and VM registers. The bytecode interpreter is basically the instruction decoder with a large conditional branch construct (case selector statement) mapping an operational code onto operational code statements, mostly consisting of stack manipulations. Due to the consecutive numbering of operational codes in the range {0,1,..,*opcodemax*-1}, a direct branch table can be used, providing an instruction decoder with a constant run-time, which is important for real-time scheduling. Commonly, C compilers detect this feature and create a branch (look-up) table implicitly. The instruction decoder and execution unit are automatically created from the perfect hash table with a switch-case construct. The operational statements are macro definitions provided by the programmer in a separate header file. Computed *goto* statements can be created, alternatively. But not all compilers, especially commercial versions tailored for embedded systems, support computed go-to statements (basically only supported by GNU compilers). The VM bytecode interpreter is fully binary compatible among different software versions as long as the same word set is used. Any changes to the core word set (number of ops or names) invalidate binary compatibility, which is the reason for bundling the VM execution with the compiler.

II. Tiny Machine Learning and Digital Signal Processing

Data-driven modelling is used in a wide range of classification and regression applications. Often, data-driven trained models (predictor functions) are fully trained before being used (application). Commonly, the model parameters as well as the variables are represented by floating-point data types and processed by floating-point hardware arithmetic, which is not available on low-resource microcontrollers. Basically, it is possible to use fixed-point arithmetic (e.g., with 16 bit encoded values), at least for classification tasks. Distributed machine learning is a specific class of ensemble learning based on the divide-and-conquer principle. Each node provides a local state estimation or classification based solely on local data, which is then globally fusioned to a global state. Assuming such an architecture, predictive classification (and possibly regression) is appealing for implementation on the sensor node level and directly processed by the microcontroller or FPGA processing unit, a concept known as "tiny machine learning." One prominent sub-class of predictive data-driven models are Artificial Neural Networks (ANN), which are basically non-linear function graphs.

An ANN can be organized in layers, and each layer consists of a given number of functional nodes (neurons). Each functional node performs a data fusion by summing the products of all input variables $\boldsymbol{x}$ (vector) with a weight parameter vector $\boldsymbol{w}$. Finally, the resulting scalar value t is passed to a commonly non-linear transfer function $g(t)$, which provides the node's output. For the computation of one node, vector operations are required.

To compute (apply) an ANN, only some specific vector arithmetic operations and a unified vector and matrix data structure are required. A challenge is the reduction in value of resolution and precision. ANNs are typically trained using floating-point arithmetic (at least with a single 32-bit precision).The VM addressed in this work supports only 16- and 32-bit integer arithmetic. The transformation of already trained networks into integer interval arithmetic requires additional scaling vectors and scaling operations. Each ANN can be functionally decomposed into vector operations. All functions f_i (representing one layer) and the output function g use matrix and vector operations, which can be implemented in software as well as hardware and computed directly with integer arithmetic. Only the activation (transfer) functions (e.g., sigmoid or soft-max) require approximated fixed-point (integer) implementations of the real-valued functions, typically using a combination of piecewise multi-point regression and look-up tables. Not fully connected ANNs are computed in the same way as fully connected ANNs, but they produce sparse vectors and matrices, resulting in a large number of null (useless) operations.

The vector operations in REXA VM's hardware architecture can be parallelized, balancing resource occupation and speed. The ARM Cortex M0-M3 processors do not provide parallel vector operations (such DSP operations were added first in generation 4).

The Input-Output System (IOS) implemented in the REXA VM is similar to the widely used Foreign Function Interface (FFI) that provides unified host application integration and extends the instruction word set with bridged native C/C++ functions (or hardware extensions). Additionally, host application variables (scalar and numerical array types) can be directly accessed from VM programs. Most of the DSP and ML operations are not core part of the core REXA-VM engine. Instead, they are added on demand by the host application from customizable libraries.

A. *Signal Interface*

A sensor node processes sensor data, which is commonly sampled by the node itself using analog-digital conversion (ADC). Active measuring techniques necessitate the generation of a stimulus, which is typically controlled by the sensor node (e.g., a digital-to-analog converter (DAC)).VM programs can access the signal acquisition layer by using a signal device interface provided by the sensor node host application via the IOS. Sampled sensor data is stored in a dedicated buffer, commonly filled automatically during the sampling phase via direct memory access (DMA). The sample buffer can be directly accessed by the VM or at program level. DMA sampling with triggering, e.g., on a specific threshold level, typically utilizes a ring buffer memory architecture. Reading the sample buffer must also be done cyclically in this case, beginning at a specific top buffer position. Due to hard resource constraints, the sample buffer is also used for digital signal processing, e.g., applying filters to the data in-place.

```
const FREE 10 const SINGLE 4 const HIGH 1
FREE 1 HIGH 100 0 adc ( Start ADC )
1000 1 sampled await ( Suspend task )
<0 if error endif
var peak 0 peak !
var offset sample0 read !
var pos
1024 0 do  ( Iterate over sample buffer )
  offset @ samples read
  dup peak @ > if peak ! i pos ! else drop endif
  offset @ 1 + 1024 mod offset !
loop
." Peak: " peak @ ." at " pos @ . cr
```

Ex. 1. Synchronous AD conversion with post processing

B. *Digital Signal Processing*

The set of DSP operations is provided via the FIOS layer API and can be extended by the host application. Only fixed-point integer arithmetic is supported. The input and output scaling of arithmetic and numerical functions is fixed. Basic operations required for typical signal processing and analysis tasks are provided, like scalable trigonometric functions, hull and filter functions.

Trigonometric functions and functions composed of trigonometric functions are implemented with segmented linear and non-linear look-up tables. For example, the error of the discrete sigmoid function is always less than 1%, while only requiring 30 bytes of LUT space and less than 10 unit operations, as shown in Alg. 1. These software functions can be immediately implemented in hardware, too. The LUTs are computed with Alg. 2.

```
static ub1 sglut13[] = { <24 values> };
static ub1 sglut310[] = { <6 elements> };
// y scale 1:1000 [0,1], x scale 1:1000
sb2 fpsigmoid(sb2 x) {
  sb2 y;
  ub1 mirror=x<0?1:0;
  if (mirror) x=-x;
  if (x>=10000) return mirror?0:1000;
  if (x<=1000) {
    y = 500+(((x*231)/1000));
    return mirror?1000-y:y;
  } else if (x<3000) {
    ub2 i10 = ((fplog10((x/5)|0)/2))-65;
    y = ((sb2)sglut13[i10])+731;
    return mirror?1000-y:y;
  } else {
    ub2 i10 = ((fplog10((x/10)|0)/10))-14;
    y = ((sb2)sglut310[i10])+952;
    return mirror?1000-y:y;
  }
  return 0;
}
static ub1 log10lut[] = { <100 values> }
// x-scale is 1:10 and log10-scale is 1:100
sb2 fplog10(sb2 x) {
  sb2 shift=0;
  while (x>=100) { shift++; x/=10; };
  return shift*100+(sb2)log10lut[x-10];
}
```

Alg. 1. Range-segmented and LUT-based implementation of the sigmoid function with less than 1% approximation error (using approximated LUT-based log10 function)

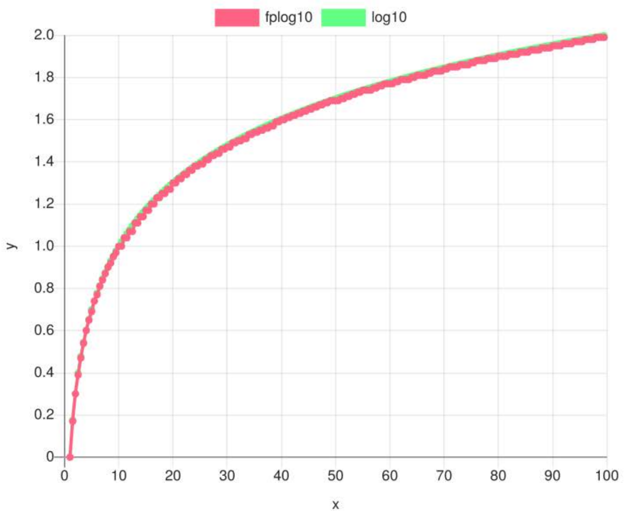
The LUT tables can be computed as follows:

$$\log_{10}\text{lut} = \left\{ int\left(\log_{10}\left(\frac{i}{10}\right)100 \right) : i \in \mathbb{I}, 0 \le i \le 99 \right\} \quad (1)$$

The *fpsigmoid* function LUTs are computed iteratively using the *fplog10* function, described by the following pseudo code algorithm Alg. 2 (accuracy is plotted in Fig. 3):

```
sglut13 := []
for x=1 to 2.95 step 0.05 do
  i10 := int(fplog10(int(x*1000/5))/2)-65
  if sglut13[i10] = undefined then
    sglut13[i10] := int(sigmoid(x)*1000)-731
  endif
done
sglut310 := []
for x=3 to 9.9 step 0.1 do
  i10 := int(fplog10(int(x*1000/10))/10)-14
  if sglut310[i10] = undefined then
    sglut310[i10] := int(sigmoid(x)*1000)-952
  endif
done
```

Alg. 2. Computation of the LUTs for the fixed-point sigmoid function



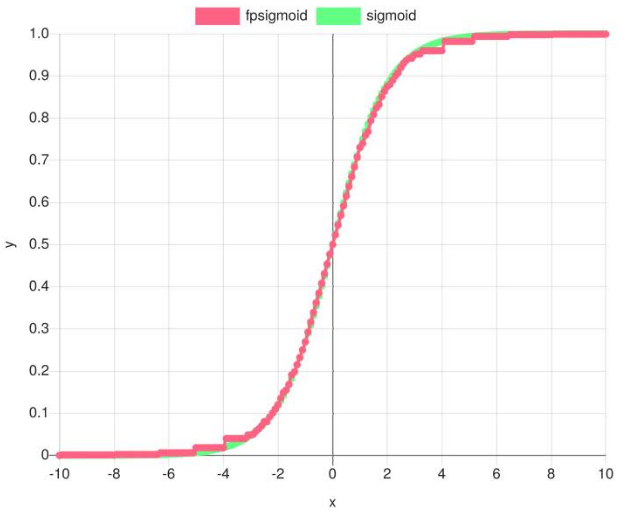


Fig. 3. Accuracy of fixed-point approximations for *log10* and *sigmoid* functions

C. *Artificial Neural Network (ANN)*

An ANN consists of two parts:

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

For the sake of simplicity, fully connected networks are assumed, but any irregular network structure is a sub-set of a fully connected structure and can be used with the following operational architectures, too. In contrast to common ANN software frameworks, the REXA VM provides only core vector operations. The parameter data is embedded in a code frame by using the initialized *array* constructor. Both parameter and input/output data can be stored in the program code frame, shown in the next section.

D. *Vector Operations*

The core set of vector operations provided by the REXA VM supporting integer arithmetic ANN computations can be summarized to:

1. Element-wise vector operations (e.g., *vecmul*: *op1vec op2vec dstvec scalevec*);
2. Dot-product operation performing a sum of product data fusion (*vecprod*: *veca vecb scale* → number);
3. A folding operation for node layer computations (*vecfold*: *invec wgtvec outvec scalevec*)
4. A mapping operation applying a function elementwise (*vecmap*: *srcvec dstvec func scalvec*)
5. And a generic scaling operations (*vecscale*: *srcvec dstvec scalevec*).

Vector operations are scaled using supplied scaling vectors (*scalevec*). Vector operations always operate on single data words (16 bit), but internally 32 bit arithmetic is used to avoid overflows. To scale to signed 16 bit integer, some of the operations use a scale factor or scale factor vector (negative scale values reduce, positive expand the values by the scale factor) to avoid overflows or underflows in following computations, similar to scaled tensors in [4,12]. There are vector loading, scaling, combination, and mapping functions, which provide basic vector ANN functions operating on embedded or external array data.

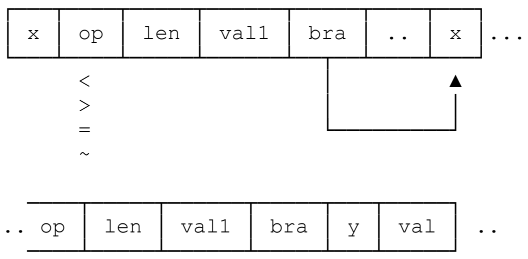
The operations are defined by the following formulas:

$$
\begin{aligned}
&vecmul\left(\vec{a},\vec{b}\right)=\left(a_1\cdot b_1,a_2\cdot b_2,..,a_n\cdot b_n\right)^T\\
&dotprod\left(\vec{a},\vec{b}\right)=\sum_{i=1}^{n}a_i\cdot b_i\\
&fold\left(\vec{a},\hat{c}\right)=\left(\sum_{i=1}^{n}a_i\cdot c_{i,1},\sum_{i=1}^{n}a_i\cdot c_{i,2},..,\sum_{i=1}^{n}a_i\cdot c_{i,n}\right)^T\\
&map\left(\vec{a},f\right)=\left(f\left(a_1\right),f\left(a_2\right),..,f\left(a_n\right)\right)^T\\
&n=\left|\vec{a}\right|=\left|\vec{b}\right|,\left|c\right|=n\cdot m
\end{aligned}
\qquad (2)
$$

E. *Decision Trees*

Decision trees, as lightweight predictor models well suited for tiny embedded systems, can be efficiently stored in Linear Search Tables (LST), as introduced earlier for compiler parsing.

Decision trees consist of nodes associated with input variables x_j or output variables y_k (and specific outcomes of a prediction). Directed edges connecting nodes are functional evaluations of a node variable.



Def. 1. Format of a Linear Search Tree (LST) implementing a decision tree

There are three basic operations: Binary relation (</>), equality (=), and nearest value approximation (≈). The data format is shown in Def. 1. Each slide starts with the input variable to be evaluated (or target for output), the operation applied to choices, a field specifying the number of choices, and value-branch pairs.

III. Use-Case: Material-integreated Structural Health Monitoring

The following complete example Ex. 2 code shows how simple it can be to implement ANNs in REXA Forth, by implementing a three layer network with [14, 8, 2] neurons (based on [13]). The 14 input features are computed using a signal hull analysis (maximum height, width, and time position) of sensor data from multi-path GUW measurements. The signal hull computation was originally performed by a Hilbert transformation, later replaced by a rectifying function and a first order recurrent low-pass filter. The network parameters (lines 3-27) are embedded in the code and are used by the network forward activation function *forward* (lines 31-47). The computations are performed by using the pre-defined universal vector operations, introduced in the previous section. The vector operations determine the size parameters of the vectors (or matrix) automatically. Impressive performance results are presented in Sec. V. The ANN feature vector is computed at run-time by multi-path GUW signal sampling (6 transducers) and simple signal analysis (lines 49-57). In line 50 the pitch signal waveform generator applied to one transducer is started (DAC), and in lines 51 the ADC measurement is started (triggered by the DAC generator). The *await* suspends the task execution until the ADC conversion is done. The results of the signal analysis (hull computation using rectification and applying a low-pass filter) are stored in the ANN input feature vector (*input*). The compiled program code requires about 800 bytes only.

```
 1 ( Signed 16 bit integer type arrays )
 2 ( Input Layer )
 3 array input  14
 4 array biasI { 1 2 .. 14 }
 5 array wghtI { 1 2 .. 14 }
 6 array scaleI { 1 2 .. 14 }
 7 array activI 14
 8
 9 ( Hidden Layer )
10 array wghtH1 {
11  1 2 .. 14 ( Neuron 1 )
```

```
  1 2 .. 14 ( Neuron 2 )
  ...
  1 2 .. 14 ( Neuron 8 )
}
array biasH1 { 1 2 .. 8 }
array scaleH1 { 1 2 .. 8 }
array activH1 8

( Output Layer )
array wghtO {
  1 2 .. 8
  1 2 .. 8
}
array biasO { 1 2 }
array scaleO { 1 2 }
array output 2


( Forward activiation of network )
: forward
  ( Evaluate input layer --   )
  input wghtI actI scaleI vecmul
  ( Add bias )
  actI biasI actI 0 vecadd
  ( Apply activation function w/o scaling )
  actI acI $ sigmoid 0 vecmap
  ( Compute hidden layer activations )
  actI wghtH1 activH1 scaleH1 vecfold
  activH1 biasH1 activH1 0 vecadd
  ( Apply activation function w/o scaling )
  activH1 activH1 $ sigmoid 0 vecmap
  ( Compute output layer activations )
  activH1 wghtO output scaleO vecfold
  output bias output 0 vecadd
  output output $ sigmoid 0 vecmap
;
( Start path measurements and feature extr. )
6 0 do
  CHIRP 10 2 1000 i dac ( Start DAC )
  EXTTRIG 1 HIGHGAIN 1000 i adc ( Start ADC )
  1000 1 sampled await ( Suspend task )
  sample 0 1024 vecabs
  sample 0 1024 10 lowp
  sample 0 1024 vecmax ( -- index val )
  input i 2 * cell+ !
  input i 2 * 1 + cell+ !
loop
forward
output vecprint cr
( Done )
```

Ex. 2. The example ANN consists of 14 input variables and neurons, one hidden layer of 8 neurons, and two output neurons. The ANN is implemented entirely in one code frame (about 400 bytes). The model parameter values are only for illustration.

The initialized vectors are stored in-place in the code frame, the non-initialized vectors are stored at the end of the code frame (extending the code frame by the compiler). If there is an additional update training function adapting weight and bias parameters, the code frame containing the ANN can be decompiled to text and send back to the source or any other device for application.

IV. Evaluation

The main advantage of the proposed VM architecture is the capability to create the main and crucial parts of the VM using code generators and adapt the VM architecture to specific applications and host architectures. All data (and code) memory is allocated statically at compile time. There is no dynamic memory management requirement. The compiler works in-place, i.e., it compiles source text stored in free regions of the CS directly in-place into bytecode (no additional storage space is required during compilation). Although the data, return, and loop stacks DS, RS, and FS, respectively, can be small, multi-threading and multi-tasking increase the stack storage requirements by the number of maximally supported threads and scheduled tasks.

Using the widely deployed 32 bit STM32 ARM Cortex M0 microcontrollers, a typical REXA VM implementation with CS=1024, DS=256, RS=128, FS=64 cells, and 101 Words, requires about 8 kB RAM and 8 kB ROM resources (not included IOS attached data and code). The Tiny ML code for ANNs requires additionally about 500 Bytes RAM and 1 kB ROM. The basic execution speed of the VM is about 14 KIPS / MHz clock frequency, i.e., 70 clock cycles and ARM machine code instructions are required for the execution of one VM bytecode instruction. The average forward computation time for an ANN with 24 neurons (see use-case) requires about 16 ms / MHz. The compiler can compile about 2000 IPS / MHZ. The complete code example from the use-case section consists of about 500 words requiring about 250 ms / MHz compile time.

The hardware resources required for typical REXA VM configurations (CS=4096, DS=1024, SS/RS=32, Words=84) for a SRAM-based FPGA XC3S500e is about 2000/4500 digital logic slices with a block RAM occupation of 9/20, showing the suitability of the REXA VM architecture for hardware implementations, too.

The DSP/ANN module of the REXA VM was tested with different ANN configurations, with 3-5 layers, and 2-64 neurons per layer. The measured results for the ARM Cortex M0, STM32 L031 and F103 microcontrollers are shown in Fig. 4 and Tab. 1 (larger networks can only be handled by the F103 due to RAM allocation).

Layers	Neurons	Code [Bytes]	Forward Time [ms/MHz]
[2,3,1]	6	237	7.7
[4,3,2]	9	281	8.2
[4,6,2]	12	336	9.1
[4,8,2]	14	372	11.2
[4,8,4]	16	416	10.5
[4,8,8,2]	22	601	14.4
[4,8,8,4]	24	645	16.6
[4,8,8,8,4]	32	874	21.0
[4,32,2]	38	804	17.1
[8,32,32,8]	80	3813	43.5
[8,64,32,8]	112	6566	58.3

Tab. 1. Normalized ANN results on STM32 platform depending on the network Configuartion (nodes per layer)

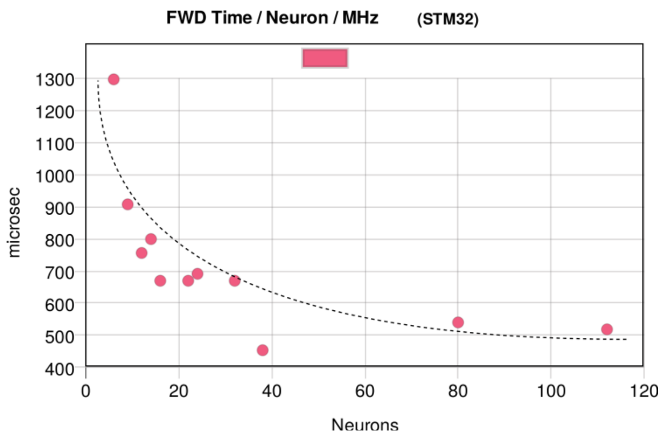


Fig. 4. ANN forward computation times for one neuron and per MHZ clock frequency (different ANN architectures with 3-5 layers and 2-64 neurons per layer)

The code size includes the ANN parameter data, input and output vectors, and the forward computation function. The code size ranges from 200 to 6K bytes for 6-100 neurons. As shown in Fig. 4, the computation time for one neuron decreases with increasing network size. This is a result of the VM execution overhead that dominates for very small network sizes. The average computation time is about 700 µs / neuron / MHz for the ARM Cortex STM32 microcontrollers. Note that the ARM processors underperform compared with modern Intel x86/x64 processors (but still have an efficiency ratio of 1:100 if power and chip area are considered). Typical computation times for medium sized networks are in the millisecond range, fully suitable for on-node classification.

V. Conclusions and Outlook

This work presented a virtualization layer (VM) for tiny embedded systems with very low resources, addressing the processing of ML models on-device in particular. Currently, only already trained ML models can be processed on the device and require a model transformation with scaling and interval integer arithmetic. The VM provides optimized core functions to compute ANN and decision tree models. The ML model is provided by text code with embedded data. In the future, initial and update training should be provided by the VM layer, too, although training algorithms can be directly implemented on programming level. ML models are provided as program code in text format with directly embedded initialized in-place data (model parameters), which can be modified at run-time. The stack VM is highly customizable and extensible (ISA, code, data, stack sizes), the integrated text-to-bytecode compiler is sufficiently fast. An optional bytecode-to-text decompiler enables mobile code and mobile ML models. Parts of the VM program code are created by parametrizable code generators supporting the domain- and application-specific optimization of the VM. The ISA can be extended by the IOS, providing programming level access to host application functions and data, e.g., for accessing ADC and DAC devices.

References

[1] S. Guo, Q. Zhou, Machine Learning on Commodity Tiny Devices, Taylor & Francis, 2023

[2] X. Wang, M. Magno, L. Cavigelli, and L. Benini, arXiv:1911.03314v3, 2022

[3] P. P. Ray, A review on TinyML: State-of-the-art and prospects, Journal of King Saud University-Computer and Information Sciences, 2021

[4] R. Banner, I. Hubara, E. Hoffer, D. Soudry, Scalable Methods for 8-bit Training of Neural Networks, arXiv:1805.11046, 2018

[5] N. N. Alajlan, D. M. Ibrahim, TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications, micromechanics, vol. 13, no. 851, 2022.

[6] Jain, S. Giraldo, J. D. Roose, B. B. Linyan Mei, M. Verhelst, TinyVers: A Tiny Versatile System-on-chip with State-Retentive eMRAM for ML Inference at the Extreme Edge, arXiv, 2023

[7] M. Bauer, IoT Virtualization with ML-based Information Extraction, in IEEE 7th World Forum on Internet of Things 2021, 2021

[8] J. R. Hayes, M. E. Fraeman, R. L. Williams, T. Zaremba, An architecture for the direct execution of the Forth programming language, (1987) ACM SIGARCH Computer Architecture News, 15(5), 42-49

[9] G. Heiser, The role of virtualization in embedded systems, In Proceedings of the 1st workshop on Isolation and integration in embedded systems (pp. 11-16), 2008

[10] B. Jenkins, http://burtleburtle.net/bob/hash/index.html, accessed 24.1.2023

[11] Y. Lu, B. Prabhakar,Perfect hashing for network applications, DOI:10.1109/ISIT.2006.261567, IEEE Explore, 2006

[12] A. Ghaffari, M. S. Tahaei, M. Tayaranian, M. Asgharian, Vahid, P. Nia, Is Integer Arithmetic Enough for Deep Learning Training?, Advances in Neural Information Processing Systems 35 (2022): 27402-27413

[13] S. Bosse, C. Polle, Eng. Proc. 2021, 10(1), 78; https://doi.org/10.3390/ecsa-8-11283

[14] S. Bosse, S. Bornemann, B. Lüssum, Virtualization of Tiny Embedded Systems with a robust real-time capable and extensible Stack Virtual Machine REXAVM supporting Material-integrated Intelligent Systems and Tiny Machine Learning , 2023, , https://doi.org/10.48550/arXiv.2302.09002