ALMA MATER STUDIORUM

Università di Bologna

FACOLTA' INGEGNERIA ELETTRONICA

Training Recurrent Neural Networks on a RISC-V Multi-Core MicroController

Candidato:
Francesco Conoscenti

Relatore:
Prof. Luca Benini

Correlatore:
Dott. Manuele Rusci

Contents

List of Figures

1 Introduction

Artificial intelligence has become one of the main methods to analyze Big Data. This data comes especially from the world around us and to catch it are used sensors or a group of sensors and actuators wirelessly connected, IoT.
Due to the sensors' technical limitations and the computational complexity of Deep Learning models, the majority of network operations are usually executed on Cloud, over high-performance servers. With this approach, latency and energy consumption for data transfer are relevant for the nodes, which are most of the time an Embedded system. To process locally the just sensed data with AI there is the needing to push Neural Network directly on the edge sensors of the IoT nets.
To achieve On Device Learning that permits processing data in local and learning from them, DNN models must be implemented over edge nodes. By far only the Inference has been implemented on the leaf sensors, but also the Backpropagation should be included to make sensor able to learn. This is even more challenging for the existing architectures because of the Computation and the Memory requirements. A trend Architecture named PULP has been chosen.

PULP is a computational platform for scalable energy efficient edge computing based on a cluster of RISC-V cores. It responds to the demand for platforms with high-computational power in environments with a low energy budget. It has a rich set of peripherals and an Open Source license.
To enable On Device Learning over PULP the library PULP-TrainLib is developed. It includes a set of performance-tunable DNN layer primitives to enable DNN training on ultra-low-power devices.
One of the state of the art DL model, that is not present in PULP-TrainLib yet, is RNN. RNN uses recurrent connections and parameters are shared across different timesteps and positions of the sequence. RNN can be treated as sequential models, which holds a certain amount of memory of the previous states, for this, it is employed to handle sequential data, such as audio or text.

The final goal is to move RNN models on the edge nodes, where the data is being directly collected, performing Forward and Backward algorithms to learn from the sensed data.

This work focus on the workload analysis of RNN & Attention. Computational and Memory evaluations are conducted over these 2 algorithms to understand the possible implementation over PULP. Consequently, the RNN primitives are developed on PULP. Finally, a test is conducted over it to measure the performances achieved. The results of the test, on RNN model, imagining to run on Vega SoC, are:
• Running Time:
  - Forward is less than 0,5 ms
  - Backward is less than 1 ms.
• Memory occupations
  - Forward around 80 KB
  - Backward around 160 KB
The performances improve with bigger net dimensions and on multiple cores, achieving speed up close to theoretical values.
The results obtained from the test highlight the possible implementation of the model on real edge devices.

2 Time Series Analisys with RNN & Attention

2.1 Artificial Intelligence, Machine Learning and Deep Learning

2.1.1 Artificial Intelligence and Machine Learning

Artificial intelligence (AI) is a family of processing techniques focusing on performing human-like tasks. The core of AI is making a machine learn from experience, processing large amounts of information and recognizing patterns in the data.

Machine learning (ML) is a branch of AI targeting a form of computer-aided applied statistics, focusing on the use of computers to statistically estimate complicated functions from large data, rather than proving confidence intervals to determine specific properties of the data.

An algorithm is said to learn from experience, if its performance at a certain tasks, measured by some metrics, improves with experience.
Machine learning tasks are usually described in terms of how the machine learning system should process an example. The classes of problems solved by ML tasks is wide; pattern recognition in data, density estimation, generation of new samples from a known set of data. In the following work, we will focus on pattern recognition. To classify ML tasks, is distinguished the strategy chosen to learn from the data.
Supervised Learning approximate the function to be learned, by providing to the ML model a dataset of examples, provided with output data labels (e.g. class labels or function values). The labels represent the classes to be learned and are used to reduce uncertainty.
Unsupervised Learning provides to the ML model a dataset on unlabeled examples, from which the model can define some of the properties of the data to reduce uncertainty, in an implicit manner.
In the following, we will focus on Supervised Learning models.

2.1.2 Deep Learning

Deep Learning (DL) is a specific branch of ML which employs Deep Neural Networks (DNNs) to analyze data. DNNs structure is composed by successive layers, each one breaking down the processing with a specific operation. Common DNN layers are Fully-Connected, Convolutional, Batch Normalization, etc.
The basic element is the Artificial Neuron, performing the operation:

$$y = F(W \cdot x + b)$$

where y is output activation, x the input activation, w the matrix of the weights, b the bias. F represents the nonlinear activation function and determines the kind of response of the neuron to the learning process. Common examples of activation functions are Sigmoid, ReLU and tanh. The peculiar feature of DNNs is that their parameters (set of weights and biases) are not set by the designer, but their values are determined by the training process, which is performed by means of the backpropagation algorithm. Training requires a loss function (Mean Squared Error, Cross Entropy) to determine the prediction error and an optimizer function (Stochastic Gradient Descent or SGD, ADAM, BPTT) to perform weight update and backpropagate the error.

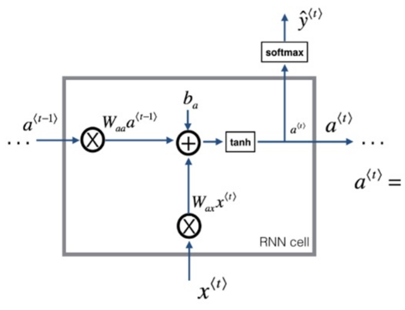
The learning process is divided into three main steps: forward or inference, backward or back-propagation and weight update. By carrying out these steps, the DNN model can improve its performance (over a certain labeled dataset) reaching a certain level of accuracy.
Training can be performed in batches, that consists in backpropagate, through the DNN model, N samples in a batch, instead of backpropagating a single input at a time. This approach speeds up the training, since the data can be efficiently regrouped to perform faster multiplications.
Online Learning approach consist on a single data backpropagated each time.

To deploy a DNN model, at first, the classifier is trained on a server, optimizing its performance up to a certain level of accuracy. Secondly, the DNN's weights (or parameters) are frozen, and the model is deployed using target-specific optimizations (DNN quantization, pruning, etc.).

2.2 RNN

Recurrent Neural Networks (RNNs) are DNN models that use recurrent connections.
The parameters of RNN models are recurrent, that means they are shared across different time steps and positions of the sequence.
This type of models can be treated as sequential, which hold a certain amount of memory of the previous states. For this, many RNN models are employed to handle sequential data, in tasks like Sequence-to-Sequence (language translation, signal filtering, . . .).

The operations of a single RNN cell are:


$\hat{y}^{\langle t \rangle}$
softmax
$a^{\langle t-1 \rangle}$
$W_{aa}a^{\langle t-1 \rangle}$
b_a
tanh
$a^{\langle t \rangle}$
$a^{\langle t \rangle}$
$W_{ax}x^{\langle t \rangle}$
RNN cell
$x^{\langle t \rangle}$

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

- Matrix Mutliplication
  -weights x inputs
  -weights x previous state
- Sum of resulting vectors
- Hyperbolic tangent

Figure 1: Logical representation of a RNN neuron [25]

A RNN model is created by applying the same set of weights recursively over a differentiable graph-like structure. The state h is computed with a recursive definition since the definition of h at time t refers back to the same definition at time t − 1 as:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

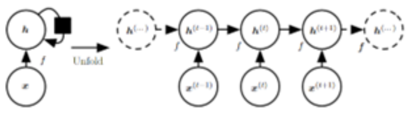


Figure 2: Process of Unfolding [26]

RNN models typically learn to use h(t) as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to the time instant t. Furthermore, unfolding parameter sharing is better than using different parameters per position because you have less parameters to estimate and generalize to various length.
We can represent the unfolded recurrence after t steps as a function g(t):

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, ..., x^{(2)}, x^{(1)}) = f(h^{(t-1)}, x^{(t)}; \theta)$$

The function g(t) takes the whole past sequence $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, .., x^{(2)}, x^{(1))}$ as input and produces the current state, but the unfolded recurrent structure allows to factorize g(t) into recursive application of a function f.

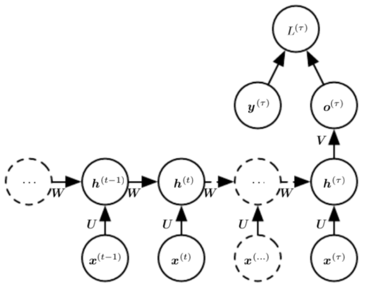


Figure 3: RNN hidden2hidden single output. It reads a sequence and produces one output [26]

Back-propagation through a single neuron start from the loss function and calculate all the partial derivative of the weight matrices (*Wxh, Wah, Wao*) and bias vectors (*bh, bo*) The single mathematical steps are reported in the appendix.
The algorithm is called Back-Propagation Through Time (BPTT), consist in the application of chain-rule on the unrolled graph for parameters of U, V, W, b and c as well as the sequence of nodes indexed by t for $x(t), h(t), o(t)$ and $L(t)$.
To update the weights and biases, we need to calculate the sum of each partial derivative $\partial Wao, \partial Wah, \partial Wxh$, at every time step t, because these parameters are shared across during forward propagation.

2.2.1 RNN Problems & Solutions

BPTT lead to some Gradient problems. In a simple linear model, the gradient grows exponentially if the matrix $\|Wrec\| > 1$, this issue is known as Exploding Gradient. On the contrary if the gradient shrinks exponentially $\|Wrec\| < 1$, it is referred to Vanishing Gradient Problem. A possible solution can be the Truncated BPTT. During the backpropagation process, we run forward and backward steps through a window called a “chunk”, of a specific size instead of the entire sequence.
Another possible solution is Gradient Clipping, method which defines a threshold, the gradient is clipped in case it overflows during the steps of the backpropagation. The long-term dependency challenge motivates various solutions such as Echo State Networks, Leaky Units and the LSTM.

2.2.2 Bidirectional RNNs

Bidirectional RNNs combine an RNN that moves forward through time beginning from the start of the sequence with another RNN that moves backward through time beginning from the end of the sequence. This architecture allows the model to have backward and forward information about the input sequence at every time step, infact the BRNN can be trained without the limitation of using input information just up to a preset future frame.

2.2.3 LSTM & GRU

Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) were created as the solution to short term memory. They have internal mechanisms called gates that can regulate the flow of information.These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences and forget non relevant data to make predictions.
LSTM has 3 gates: forget, output and input and cell state. It processes data passing on information as it propagates forward. The core concept of LSTM are the cell state, It transfers relative information all the way down the sequence chain. So, even information from the earlier time steps can make it's way to later time steps. As the cell state goes trough, information get's added or removed to the cell state via gates. The gates are different neural networks that can learn and decide which information is allowed on the cell state.
GRU presents 2 gates: update and reset. It is like a LSTM but has fewer parameters, as it lacks an output gate. GRU got rid of the cell state and use the hidden state to transfer information.

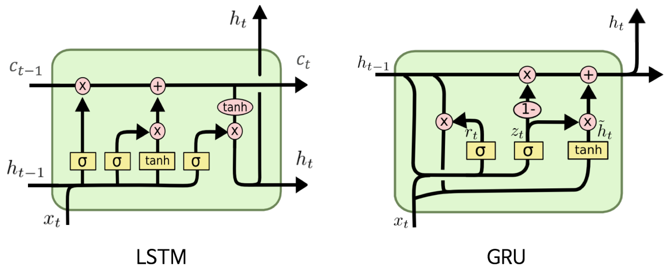


Figure 4: LSTM & GRU Block Diagram[28]

2.3 Attention

In ML it's called "Attention" any layer used to emphasize "important" parts of data and depress "unimportant" ones, imitating the cognitive mechanism of human attention. The tasks which Attention is designed for are: reading comprehension, abstractive summarization, textual entailment and learning task-independent sentence representations.
An attention function can be described as mapping a query and a key-value pairs to an output. The output is computed as a weighted sum of the values. The weight assigned to each value is computed by a similarity function of the query with the key.
For example, when you search for videos on a search engine, it will map your query (text in the search bar) against a set of keys (video title, description, etc.) associated with candidate videos in their database, then present you the best matched videos (values). The current Query is matched with the memory, in search for similar Keys. A softmax function is used to calculate a

distribution of the relevant words found by this matching operation. This distribution is called "Attention Score".
There are three separate Linear layers for generating Query, Key, and Value. Each Linear layer has its own weights. The input is passed through these Linear layers to produce the K, and V matrices. In Attention the Query is generated from the target sequence, in Self Attention it coincides with input sequence.

Q = XWquery, K = XWkey, V = XWvalue

where Wquery , Wkey and Wvalue are all matrices of size E x P (space of embeddings E, projection space P)

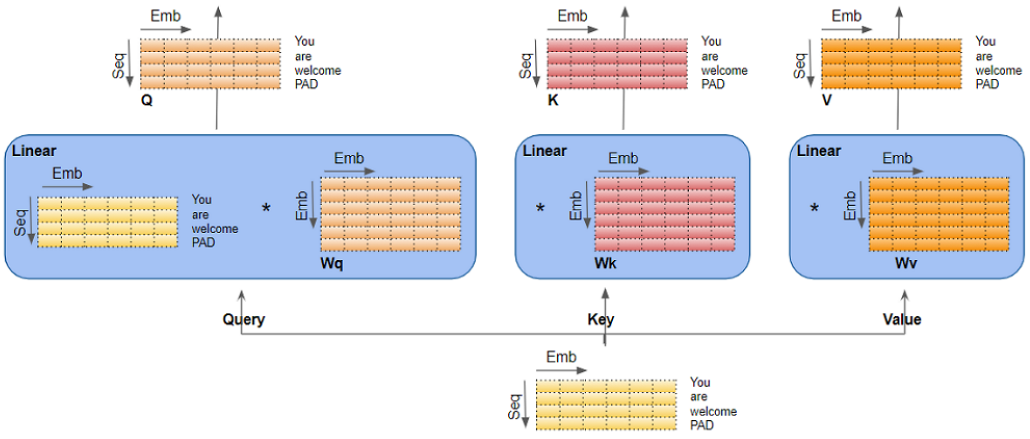


Figure 5: Attention Input passes in the Linear Layer to generate Key, Query and Value [23]

Input and weights are multiplied to increase the possibility for each input token to attend to other tokens in the sequence, instead of just itself. This results in a better representation of the input vector and converts of the input vector into a space with a compatible dimensions. Note that the weights of the linear projection are learnable, without manual setting.

$$Attention(Q,K,V) = AV = Softmax\left(\frac{QK^T}{\sqrt{P}}\right)V$$

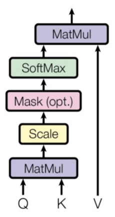


Figure 6: Attention operation [24]

To compute the Attention Score, the Q, K, and V matrices are used. Masking is used to

avoid including Padding values in the computation of the Attention Score. The first operation is a self-similarity operation between the 2 matrices (Query, Key), that consist in a scaled dot product.
The result, called "Attention Scores", is a square matrix that highlights the relation between the words in the sentence. This result is then normalized with a softmax, obtaining a mask that represent a probability distribution over the Keys. The result is finally multiplied with the embeddings (stored in the Value matrix).

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^o$$
$$where \quad head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

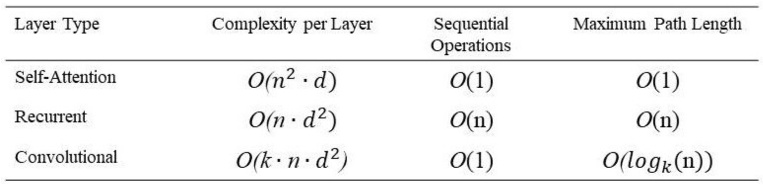
2.3.1 Attention & Self Attention

If the keys, values, and queries are generated from the same sequence, then we call it Self-Attention. Attention is converts the input sequence into a target sequence that may represent the "focus" and "most valuable" parts of the sequence in a more significant form. Attention mechanisms allow the output sequence to focus attention on the input one, while the Self-Attention model allows inputs to interact with each other, relating different positions of a single sequence.

2.3.2 Multihead Attention

Multihead Attention linearly project the Query, Key and Value matrices to a set of dq, dk and dv submatrices of the dimension in which the subspaces are projected h times with different learned linear projections.
On each of these projected versions of queries, keys and values it is then performed the attention function in parallel, yielding dv-dimensional output values. These are concatenated and once again projected, with a dot product with Wo, to obtain the final values.
Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions. With a single attention head, averaging inhibits this flow of information from different sources.

2.3.3 Confronting Self Attention with other learning algorithms



Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(\mathrm{n})$	$O(\mathrm{n})$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(\mathrm{n}))$

Figure 7: Table to confront the computation of the algorithms [25]

Self-Attention layer connects all positions with a constant number of sequentially executed operations, whereas a recurrent layer requires *O(n)* sequential operations. The RNN and convolution

compress the data and calculate a positional similarity confronting with the previous sequential inputs. Aligning the positions to steps in computation time, they generate a sequence of hidden states, this inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.
Instead, Attention uses content-based querying, predictions are made based on semantic, and can be parallelized. In terms of computational complexity, self-attention layers are faster than recurrent layers when the sequence length n is smaller than the representation dimensionality d. To improve computational performance for tasks involving very long sequences, self-attention could be restricted to considering only a neighborhood of size r in the input sequence, centered around the respective output position. This would increase the maximum path length to *O(n/r)*.

2.4 Evaluation and computational analysis of RNN and Attention

To evaluate the possible implementation of the Attention mechanism and of the RNN model on PULP, it is necessary an estimation of the computational effort and memory requirements.
The goal of this work is to minimize these values.

The results obtained by those estimation are dependent on the dimension of the matrices and on other model parameters.

The performance unit measures used in the estimation are MAC and Mem_occ.
MAC (Multiply and ACcumulate) refers to a single operation of the processor of multiplication and accumulation $a \leftarrow a + (b \times c)$. To estimate the computational effort of the algorithms this value is fundamental because it emulate an elementary operation of the processor.
Mem_occ is the measure of the size, in bytes, occupied in the memory by all the values that make up the net.

2.4.1 RNN

The hypotheses that are assumed in the following calculus are:

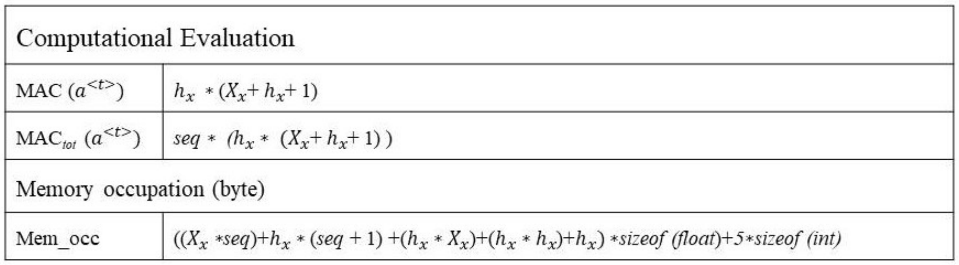
- counting the computation of the *tanh* as overhead.
- the input is considered divided in minibatches.
- the output layer with the *softmax* operation is not analyzed.
- the operations are done *In situ*, which means reusing the memory occupied by the operands to store the result.

The operation of a single Recurrent Neuron is:

$$\boxed{a^{<t>} = tanh(W_{ax}x^{<t>} + W_{aa}a^{<t-1>} + b_a)}$$

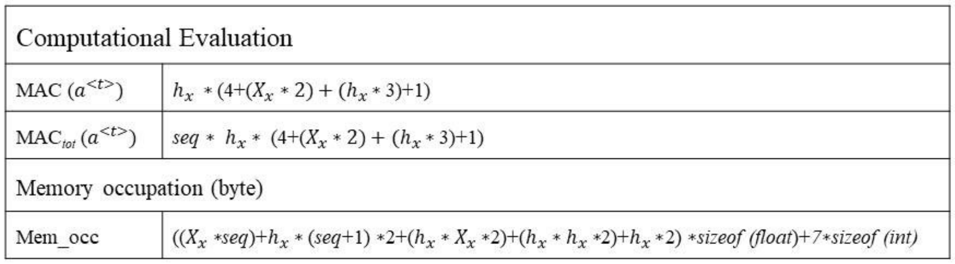
Input:	$x^{<t>}(X_s)$
Hidden state:	$a^{<t>}(h_s)$
Weight Matricies:	$W_{ax}(h_s, X_s)$, $W_{aa}(h_s, h_s)$
number of ricorsion:	seq

FORWARD



Computational Evaluation	
MAC ($a^{<t>}$)	$h_x * (X_x + h_x + 1)$
MAC$_{tot}$ ($a^{<t>}$)	$seq * (h_x * (X_x + h_x + 1))$
Memory occupation (byte)	
Mem_occ	$((X_x * seq) + h_x * (seq + 1) + (h_x * X_x) + (h_x * h_x) + h_x) * sizeof(float) + 5 * sizeof(int)$

BACKWARD



Computational Evaluation	
MAC ($a^{<t>}$)	$h_x * (4 + (X_x * 2) + (h_x * 3) + 1)$
MAC$_{tot}$ ($a^{<t>}$)	$seq * h_x * (4 + (X_x * 2) + (h_x * 3) + 1)$
Memory occupation (byte)	
Mem_occ	$((X_x * seq) + h_x * (seq+1) * 2 + (h_x * X_x * 2) + (h_x * h_x * 2) + h_x * 2) * sizeof(float) + 7 * sizeof(int)$

2.4.2 Attention

The hypotheses that are assumed in the following calculus are:

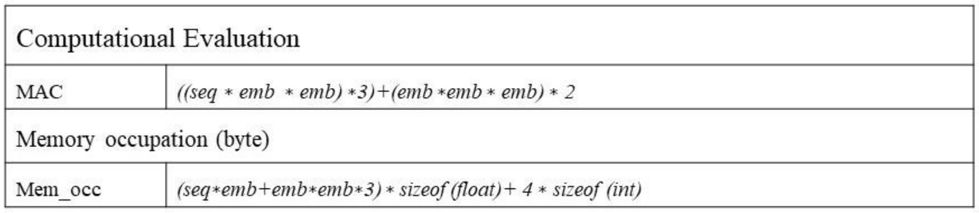
- counting the computation of the *softmax* as overhead.
- counting the computation of the Transpose algorithm as overhead.

The Attention algorithm can be expressed as:

$$\boxed{Attention(Q,K,V) = AV = Softmax\left(\frac{QK^T}{\sqrt{P}}\right)V}$$

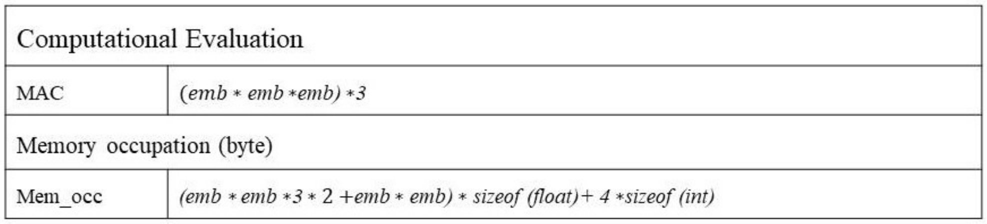
Input:	$X_i(seq, emb)$
Weight Matricies:	$W_q(emb, emb)$, $W_k(emb, emb)$, $W_v(emb, emb)$
Query, Key, Value:	$Q = W_q$ x X_i $K = W_k$ x X_i $V = W_v$ x X_i

FORWARD



Computational Evaluation	
MAC	*((seq * emb * emb) *3)+(emb *emb * emb) * 2*
Memory occupation (byte)	
Mem_occ	*(seq*emb+emb*emb*3) * sizeof (float)+ 4 * sizeof (int)*

BACKWARD



Computational Evaluation	
MAC	*(emb * emb *emb) *3*
Memory occupation (byte)	
Mem_occ	*(emb * emb *3 * 2 +emb * emb) * sizeof (float)+ 4 *sizeof (int)*

3 RNN on PULP

3.1 PULP

PULP (Parallel Ultra-Low Power) is a computational platform for scalable energy efficient edge computing based on a cluster of RISC-V cores.
The PULP project responds to the demand for platforms with high-computational power in environments with a low energy budget.
PULP enables cost-effective development and deployment of intelligent devices that capture, analyze, and classify on wide variety of rich data sources such as images or sound. This allows integration of artificial intelligence capabilities on wireless edge devices for IoT applications including image recognition, counting and recognition, and many more.

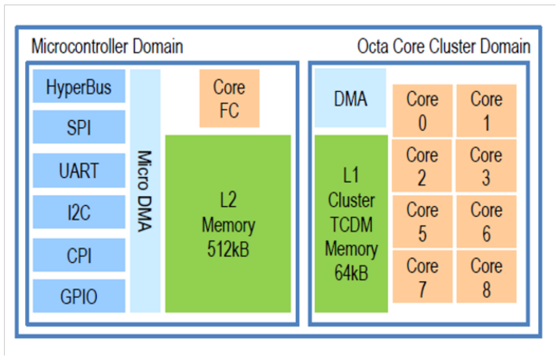


Figure 8: PULP components diagram[22]

The PULP System on Chip (SoC) instance includes a Micro-Controller Unit (MCU) for control-related tasks, and a multi-core cluster for parallel computation. The MCU features a single RISC-V core, a set of IO peripherals, and a 2 MB SRAM memory L2 accessible by the cluster side through a Direct Memory Access (DMA) engine.

The cluster includes 8 RISC-V cores sharing a 64 kB L1 memory. The cluster domain is connected to all external resources and peripherals via a peripheral interconnect. A custom extension to support multiple additional instructions that are not part of the standard RISC-V ISA is supported. The extension (Xpulp) provides DSP-like features to reduce overhead in highly uniform workloads, including post-increment load/store operations and 2-level nested hardware loops.

A commercial product developed on PULP is GAP8. It is an IoT application processor developed by Greenwave Technologies. It is built upon the open-source PULP platform, implementing an extended version of the RISC-V instruction set.

3.1.1 HW components

• Cluster of 8 OpenRISC cores that execute compute-intensive tasks in parallel. All 8 cores of the cluster share the RV32IMFCXpulp instruction set architecture.
• A Fabric Controller (FC) subsystem is used for control, communication and Security. It is a high-performance micro-controller and it uses the same Instruction Set Architecture (ISA) of the Cluster cores
• TCDM (Tightly Coupled Data Memory) is a Shared-Memory parallel programming model with 4-16 DSP cores with a shared L1 Memory. It has a number of ports equal to the number of Processing Elements (PE), allowing concurrent access to different memory locations. The L1 memory can serve all memory requests in parallel with single-cycle access latency, thanks to a low-latency logarithmic interconnect featuring a word-level interleaving scheme with round-robin arbitration, resulting into a low average contention rate
• L2 Memory accessible by all processors, divided into 4 blocks
• HyperBus Interface to connect external HyperFlash or HyperRAM
• Communication interfaces o 32 GPIO, SPI Master, SPI Slave, UART, 2 I2C, I2S, JTAG and many more.
• Multi-channel DMA allowing fast memory transfers between cores, L2 memory, and peripherals. It's connected to the TCDM with a low latency interconnect, the same used by the PEs. This reduces internal buffering when managing L1 data transfers and lightens the load of the RISC-V processor during the I/O operations
• micro-DMA capable of handling complex I/O scheme
• 2 basic timers, 4 advanced PWM timers, a real-time clock

The cluster is inactive and clock-gated at boot: a single thread runs on the Fabric Controller. After activating the cluster, a function is called on the cluster core with ID 0 (the "master" core of the cluster). The master core is the only one directly communicating with the FC, and it has to manage the parallelization of the remaining cores by dispatching the tasks. Forked executions on multiple cluster cores are synchronized with barriers, critical sections. The cluster are turned off once the tasks are finished to save energy, after transferring memory from L1 to L2.

The peripheral architecture allows for the system to be in two different operating modes:
• Slave mode: PULP acts as a multi-core accelerator of a standard host processor.
• Standalone mode: PULP detects external flash memory on the SPI master interface.

PULP is capable of switching to a sleep state with very low power consumption when not performing any task. Frequency-Locked Loop (FLL), permit each core to operate on private voltage and frequency. Furthermore, it is equipped with a Power Management Unit (PMU) to quickly switch each part of the architecture between normal and a "boost" mode.
About the data type supported, the memories are byte addressable, so every single data type whose size is a multiple of bytes can be supported, either natively if the number of bytes is less or equal than 4 or through software emulation if it is larger.
There is only a single shared instruction cache, instead of multiple private ones. This is because the cluster cores execute the same area of code, while parallelization is done on data level. By using a single instruction cache, memory access by instructions is generally reduced.

3.1.2 SW Environment

The GCC compiler used for PULP is based on GNU GCC and supports the PULP ISA, based on the RISC-V standard ISA and specific extensions, such as Xpulp version 0 to 2 and XpulpNN, which have different features and application domains.

The PULP-SDK includes a PULP platform simulator (GVSOC) and the software libraries. GVSOC is a lightweight virtual platform, simulating the GAP8 IoT application processor. This allows to test out programs without any hardware limits. It is also possible to simulate full applications with real device drivers, by using device models. Main SDK features include tools for fast native simulation (C++), instantiation and configuration (Python). Performance is measured using hardware counters, the simulation is around 1 MIPS of speed. It is functionally aligned and calibrated with the hardware with a timing accuracy in the range of 10 to 20%.
In order to execute and simulate in the PULP platform "Build automation" is the process of automating the main steps required to create a software, including compiling, assembling, linking and (possibly) testing. This is done with the command $ *make clean all run*, where: *Clean*: previous build processes (folder BUILD/), *All*: Compile the program and produce the binary code, *Run*: binary code on the target platform.

3.1.3 PMSIS

The PULP Microcontroller Software Interface Standard (PMSIS) provides the Application Programming Interface (API), the Board Support Package (BSP), and the drivers for running applications on PULP.
The PMSIS API is a set of low-level drivers which provide a common layer to upper layers to any operating systems that implement it. Together with the PMSIS BSP, it provides a full stack of drivers, allowing the development of applications that are portable across a wide range of OS.
All functions prefixed by pi_ can only be called from fabric-controller, they are by default synchronous and are blocking the caller until the operation is done. All functions prefixed by pi_cl_ can only be called from cluster.
All functions suffixed by _async are asynchronous and are not blocking the caller. The termination of such operations is managed with a pi_task_t object.
Some useful functions include:

• `int pi_cl_cluster_nb_cores()`
Return the number of cores of the cluster.

• `void pi_cl_team_fork(int nb_cores, void (*entry)) void*`
void *argFork the execution of the calling core. Calling this function will create a team of workers and call the specified entry point on each core of the team to start multi-core processing.
Parameters
nb_cores: The number of cores which will enter the entry point.
entry: The function entry point to be executed by all cores of the team.
arg: The argument of the function entry point.

• `void pi_cl_team_fork_task(struct pi_cl_team_task *fork_task)`
Fork the execution of the calling core using task. Calling this function will create a team of workers and call the specified entry point on each core of the team to start multi-core processing

with the team parameters specified in the task.
Parameters
fork_task: Task to be forked on slave cores

• `void pi_cl_team_barrier()`
Execute a barrier between all cores of the team. This will block the execution of each calling core until all cores have reached the barrier.

• `void pi_cl_team_critical_enter()`
Enter a critical section. This will block the execution of the calling core until it can execute the following section of code alone.

• `void pi_cl_team_critical_exit()`
Exit a critical section. This will exit the critical code and let other cores executing it.

3.1.4 Parallelization

PULP platform allows the parallelization of the running code, thanks to the PMSIS's functions listed above. Parallelization can be fundamental to achieve better performances because it permits each core to run a different part of the code simultaneously and not sequentially. The theoretical maximum speed up that can be reached is shown by the Amdahl's law.

$$SpeedUp = \frac{1}{(s + \frac{p}{N})}$$

s is the percentage of the code not parallelizable
p is the percentage of the code perfectly parallelizable (p = 1 - s)
N is the number of cores for parallelization

The PULP Hardware has 8 cores for parallelization.
The PMSIS library provides useful functions to perform an optimized parallelization on all the cores. In the case of matmul parallelization, it becomes:

```
pi_cl_team_fork (NUM_CORES, matmul_type, matMul_args)
```

Calling this function will create a team of workers and call the specified entry point on each core of the team to start multi-core processing. We are talking about Data level parallelism (DLP), that means that the same operation is applied to multiple data in parallel. This is different than Instruction level parallelism (ILP) which measure how many different operations are simultaneously executable.
Different matmul functions work with different parallelization algorithms[13] that are more or less efficient for different dimensions of the Matrices involved in the operation.

The tanh function has been parallelized. tanh is essential for the RNN because is done every TimeStep over all data. It's crucial that it is not too heavy in computational terms.
3 types of tanh have been developed, with different levels of optimization.

• Naïve tanh:
it uses the C function tanhf, made to compute float values. Computationally is very heavy and most of the operations of the code are because of it.

• Faster tanh:
it uses an algorithm very optimized to decrease the MAC. This has been taken from *https://github.com/pmineiro/fastapprox/tree/master/fastapprox/src.* This algorithm uses a higher approximation range to fast up the execution. It depends on certain libraries included in *tests/test_RNN* like *xmminstrin.h, fasthyperbollic.h, fastexp.h.* It uses an advanced mathematical algorithm, the fast inverse square root. The tests done confirm the correctness of the results and its efficiency in comparison to the Naive tanh.

• Parallel tanh:
Faster tanh is parallelized, splitting the input matrix into multiple blocks and process the tanh operation in parallel on these blocks. It is used a new structure to pass the data to the fork function, called tanh_arg. The fork function itself do the actual parallelization.

3.1.5 PULP TrainLib

PULP-TrainLib is the first open-source training library for RISC-V-based multicore MCUs. It includes a set of performance-tunable DNN layer primitives to enable DNN training on ultra-low-power devices.
PULP-TrainLib has features like optimizers, losses and activation functions. To enable on-device training, is also equipped with AutoTuner, a pre-deployment tool to select the fastest configuration for each DNN layer, according to the training step to be performed and the shapes of the layer tensors. To facilitate the deployment of training tasks, it has a TrainLib Deployer, that is a code generator capable of generating a project folder containing all the files and the code to run a DNN training task on PULP.
PULP-TrainLib's Autotuner finds the fastest (TileShape/MM) couple for a given layer and training step. PULP-TrainLib is designed to work with different data types. For each data format, a set of source files is designed. Furthermore, many of the functions inside PULP-TrainLib require specific data structures to work, passing the operands implicitly (as void* args) for compatibility with PULP PMSIS.

The PULP-TrainLib is organized in different files and folders.
The *tests/* folder provides useful tests to try and verify PULP-TrainLib's layers and functions. Each test can be customized according to the user specifications and profiles the execution of the layer's primitives with PULP's performance counters. The tests are organized the reference data is generated by *utils/GM.py.* In each test folder inside each test, the main code for the training steps is contained into net.c.
The *tools/* folder contains useful tools to complement the library, like the AutoTuner and the TrainLib Deployer. All the library files are included in the *lib/* folder. It contains *include/* folder containing the header files of the library, together with all of the definitions of the structures and the function primitives and the *sources/* folder containing all the implementations.

PULP-TrainLib to be used and compiled requires PULP-SDK and the RISC-V GNU GCC TOOLCHAIN. To successfully run the tests, Python 3.6 is needed, together with PyTorch 1.9.0.

Artificial Intelligence for Embedded Systems (AIfES) provides similar solutions to PULP Train-Lib. It is a platform-independent and standalone AI framework for embedded systems. It is optimized especially for the Arduino IDE. AIfES is developed in the C programming language and uses only standard libraries based on the GNU Compiler Collection (GCC). AIfES thus runs on almost any hardware from 8-bit microcontrollers over Raspberry PI to smartphones or PCs. Not only inference of FNN is possible, but also training directly in the device.

3.2 RNN primitives

In this section are reported the RNN's primitives, Forward and Backward, represented as Block Diagrams. The mathematical formulas resume step by step the algorithms presented.

Forward

$$\boxed{outData1 = W_{ax} \times Input^{<t>}}$$

$$\boxed{outData2 = W_{aa} \times hidState^{<t-1>}}$$

$$\boxed{hidState^{<t>} = tanh(outData1 + outData2)}$$

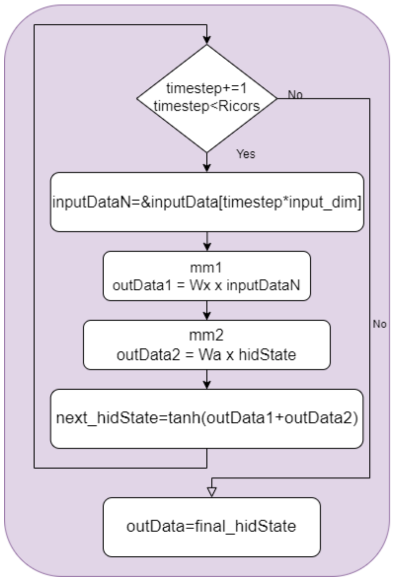

timestep+=1
timestep<Ricors
No
Yes
inputDataN=&inputData[timestep*input_dim]
mm1
outData1 = Wx x inputDataN
mm2
outData2 = Wa x hidState
No
next_hidState=tanh(outData1+outData2)
outData=final_hidState

Figure 9: Forward Block diagram

The *timestep* variable is the loop counter. The input is divided into mini-batches, each step, one is fed into the neuron. The matmuls (*mm*) are the core of the computation. Finally, the non-linear function tanh act as activation function. The output is the final hidden state.

Backward

$$\boxed{hidStateDiff^{<t>} = (1 - hidState^{<t>2}) \cdot (hidStateDiff^{<t+1>} \times Wa)}$$

$$\boxed{grad = (1 - hidState^{<t>2}) \cdot hidStateDiff^{<t+1>}}$$

$$\boxed{gradWx = grad \times Input^{<t-1>}}$$

$$\boxed{gradWa = grad \times hidState^{<t-1>}}$$

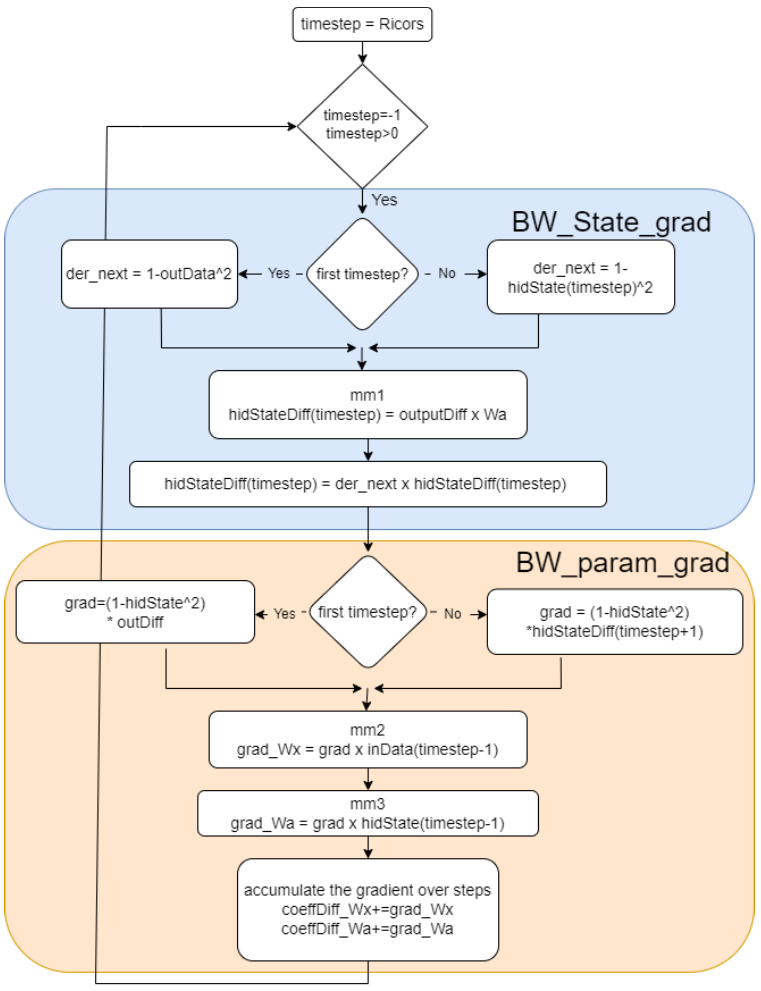

timestep = Ricors
timestep=-1
timestep>0
Yes
BW_State_grad
der_next = 1-outData^2
Yes
first timestep?
No
der_next = 1-hidState(timestep)^2
mm1
hidStateDiff(timestep) = outputDiff x Wa
hidStateDiff(timestep) = der_next x hidStateDiff(timestep)
BW_param_grad
grad=(1-hidState^2) * outDiff
Yes
first timestep?
No
grad = (1-hidState^2) *hidStateDiff(timestep+1)
mm2
grad_Wx = grad x inData(timestep-1)
mm3
grad_Wa = grad x hidState(timestep-1)
accumulate the gradient over steps
coeffDiff_Wx+=grad_Wx
coeffDiff_Wa+=grad_Wa

Figure 10: Backward Block diagram

The process of unfolding is realized, starting from the last *timestep*. The gradient is calculated with respect to the *outData* the first timestep, then with respect to the *hidden state*. Finally, the gradients of the weights are calculated and accumulated.

3.3 Implementation and GM

A GM (Golden Model) replicates the model under testing on a higher abstraction level, and generates some values that are compared with the ones generated by the testing model, to verify their correctness.
The GM used in this work was developed on Pytorch. It is a replica of the model developed using PULP primitives but is written using the Pytorch framework. Thanks to the framework precision, we can consider the results of the GM as a benchmark to validate our custom model. These resulting data are called *Golden data.*

3.3.1 Golden Models Overview

The Pythorch GM doesn't support the access to the hidden states' data in-between recursions. A second testing model has been generated using NumPy functions, which allows access to all data at every timestep. This enable a deeper and more sequential check of the data.
The two models produce equivalent final outputs, written in the same file, *RNN-data.h*, with also the hidden states, generated by the second model.
Pythorch enables an abstraction of the RNN's architecture, it works with objects and classes. Instead, in numphy there are only matrix and vector multiplications.
Below is a commented pseudo-code that shows the fundamental functions of both the models.

Pytorch Model

```
class RNN (nn.Module):

    def --init--(self, input_size, hidden_size, num_layers):
        super(RNN, self).--init--()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)


    def forward(self, x):

        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)

        out,_ = self.rnn(x, h0)  #return output of each time step
        # out: (batch_size, seq_length, hidden_size)

        out = out[:, -1, :] # I take the last time step as final output
        # out: (batch_size, hidden_size)

        return  out


net = RNN(in_size, hidden_size, num_layers) # Define and initialize net

indata = torch.div(torch.ones(batch_size,seq_length,in_size),10)  #  input data

label = torch.ones(batch_size, hidden_size) # generate label data
```

```
criterion = nn.MSELoss() # Optimizer and criterion


for i in range(1):

    output = net.forward(indata) # (batch_size x hidden_size) # forward

    loss = criterion(output, label)

    loss_meanval = 1/hidden_size # Manually compute outdiff

    output_diff = loss_meanval * 2.0 * (output - label)

    loss.backward() # Backward and show gradients


    for name, parameter in net.named_parameters():

        print(name, parameter.grad)

    print("\nInput grad is: ", indata.grad)
```

NumPy Model

```
def update_state(xk, sk, wx, wRec,bih,bhh):     #operation of a single RNN cell

    return np.tanh(np.matmul(wx,xk) + bih   +  np.matmul(wRec,sk) + bhh )


def forward_states(X, wx, wRec,bih,bhh): #calculate the next hidden state

    S = np.zeros((hidden_size, len(X[:,1])+1)) #Initialize to zero hidden state

    for k in range(0, len(X[:,1])): #Compute hidden state over the sequence

        S[:,k+1] = update_state(X[k,:], S[:,k], wx, wRec,bih,bhh)

    return S


def backward_gradient(X, S, grad_out, W): #backward algorithm

    #Initialize array that stores the gradients of the loss respect to states
    grad_over_time = np.zeros((hidden_size, nb_of_samples+1))

    grad_over_time[:,-1] = grad_out

    for k in range( X.shape[0]-1, -1, -1): #loop over all the input batches
```

```
        dnext=1-(S[:,k+1])**2

        derfin= (dnext*grad_over_time[:,k+1]).T

        wx_grad += np.dot(  derfin  , X_riga)
        #(h_size x 1) x (1 x in_size) = (h_size x in_size)

        wRec_grad += np.dot(  derfin ,  S_riga)
        #(h_size x 1) x (1 x h_size) = (h_size x h_size)

        # Compute the gradient at the output of the previous layer
        grad_over_time[:,k] =(dnext)*(np.matmul(grad_over_time[:,k+1],W[1].T))


return (wx_grad, wRec_grad), grad_over_time
```

3.3.2 Repository Structure

The whole project is distributed in many files.
The main distinction between these files is:
.c files: contain the body of the functions.
.h files: contain the headers, which are the declaration of the functions and the structure types' user definitions.

test/RNN
main.c contains the function for the configuration of the cluster, and launches the training procedure on it using *net_ step()*.

net.c contains all the important functions for running the net, including the above mentioned *net_ step()*.

net_ step() Contains 3 main functions

tensor_ init() Initializes the tensor of input, output and weights

connect_ blobs() that connects the tensor to the data type blob

train() that chooses whether to run, on PULP, the forward or backward algorithm, based on the keyword defined in *step-check.h.* The call of this functions is surrounded by the profiling functions, finally, *compare_ tensor* and *check_ tensor* are called using the output tensors as an argument.

Net.h defines important variables and functions like the checksum, *check_ tensor* and *compare_ tensor*, described below.

Read_ write_ stats.py contains a program that writes on a Calc file the performance stats of the RNN

RNN-data.h contains the data generated by the GM.

Step-check.h contains the name of the algorithm to run (define FORWARD or BACKWARD). Written by the GM and used by the *train()* function to choose which function to run.

Makefile is the configuration file containing all the User settings and flags for compiling. It sets rules to determine which parts of a program need to be recompiled. Furthermore offers a standardized way to specify dependencies and a tool to parse the dependencies and run associated build actions

test/RNN/utils
GM.py is the Golden Model written in Pytorch generating the golden data.

Dump_utils.py, profileoptimized.py, profile_sizes.py,profile_utils.py files that format and parse the output text and contain some side functions.

Lib/include
pulp_train.h contains all the inclusion of the used libraries, the standard ones, *pmsis.h, stdio.h* and also the one containing the structures and the primitives, *pulp_matmul_fp32.h, pulp_RNN.h.*

pulp_train_defines.h defines some data format and some general macros

pulp_train_utilsfp32.h contains the definition of some structures and functions, for example, blob, and all the structures used as a function argument, like *relu_args* and *matmul_arg.*

Lib/sources
pulp_RNN.c contains the body of the RNN's forward and backward algorithms, called inside the *train()* function.

pulp_train_utilsfp32.c contains the body of some useful functions, in particular the *mm_manager*, that choose the matmul type with respect to the layer and the step selected.

3.3.3 Data type and structure

Two main data representations are used in this work, and no other format is supported.

The data formats are:

fp_16: refers to the FP16 Sign-Exponent-Mantissa 1-5-10 format
fp_32: refers to the FP32 Sign-Exponent-Mantissa 1-8-23 format

The fundamental structure of the library is the "blob" structure, that wraps the tensor and link the data array and the tensor sizes to it.

```
struct blob{

    float* data;        pointer of the input data array
    float* diff;        pointer of the input diff array
    int dim;            size of the data as a 1-D array in the memory
    int W;              width of data
    int H;              height of data
    int C;              number of channels of data
    }
```

Another important structure is the one required as an argument by the *mm* (Matrix Multiplication).

```
Struct matMul_args{

    float* _restrict_ A;      pointer to the first matrix
    float* _restrict_ B;      pointer to the second matrix
    float* _restrict_ C;      pointer to the resulting matrix
    int N;                    number of lines of first matrix
    int M;                    number of columns of second matrix
    int K;                    number of columns of first  lines of second matrix
    int trans_B;              if the second matrix is transpose
    }
```

3.4 Testing Stats

In this part of the work, we check the model's stability, we compare the different results and finally analyze the computational effort when running.

3.4.1 Testing procedure

On a standard Linux shell, we run the command line *$ source /pulp/sourceme.sh* to run the simulator. The command *make clean get_golden all run* starts the GM, which generates the validity values, then runs the custom model on PULP, and finally compares the outputs. It is possible to add some settings to this command, changeable also in the *Makefile*. For example: *rm -rf BUILD* to delete the BUILD folder before running, or *NUM_CORES=N* to specify the number of cores to run the code, or *runner_args=" -trace=<PATH >:log.txt"* that allows dumping architecture events to help developers in debugging, or *runner_args="- vcd"* to generate a file which contains the most interesting signals value . Many more settings are available like net dimensions, loop optimizer or flags.

3.4.2 Check results

The functions *check_tensor()* and *compare_tensor()* are responsible to verify the correspondence between the GM and PULP model results.

• *compare_tensor()*: takes as input the pointer to the 2 different tensors and their length, to compare them. It makes a vectorized subtraction and then averages the obtained values to check if the resulting value is smaller than the ERROR_TOLERANCE defined in *net.h.* Finally print a message if the tensors are matching or not.

• *check_tensor()*: if the difference between the comparing tensors is bigger than the CHECK_TOLERANCE, prints the values of the non matching tensors.

3.4.3 Stats

Profiling functions have been developed, they extract some statistical measurements fundamental for the evaluation of the code performance. Some useful statistic measurement are the number of cycles, the number of instructions their relationship and impact on latency. To extract these values, the performance counter struct is configured with the variables to be extracted. Then the counter is stopped, reset, and started again. Then, after the function to evaluate, the counter is stopped. Finally, the variables configured are read.

```
pi_perf_conf (( 1«PI_PERF_CYCLES) (1«PI_PERF_INSTR))
pi_perf_stop()
pi_perf_reset()
pi_perf_start()

........function to evaluate.........

pi_perf_stop()

int  cycles=pi_perf_read(PI_PERF_CYCLES)
int   instructions =pi_perf_read(PI_PERF_ INSTR)
```

In this way, you can assign to a variable the value of the statistic measurement done with the function.
Then, to write in a text or Calc file these statistical variables a specific python file in the same folder has been included.
It can be run as a normal Python script on a command shell. The script reads the text file on which the RNN wrote its output and opens in write mode the Calc file to write the performance stats. Then a keyword is searched in the output file of the RNN, in this case "performance". In this way, the values of interest (stats) are found and collected in the Calc file.

4 Results

4.1 Experimental setup

IIn this section, it is analyzed the performance experimental results of the RNN in PULP.
It is highlighted the dependency of the performances on the net's parameters, especially the net's dimension and the number of cores, but also the different algorithms are chosen, matmul, tanh.
The section is developed showing the performance results over each parameter or function. The editable parameters are shown one by one and are presented the performance improvement related to them.
To analyze performances and to measure the relative improvements are chosen some quantities:

- M/C (MAC/Cycles): this metric shows how many MAC are performed per clock cycle. The number of theoretical MAC operations refers to the formulas in the second section, and represents the computational cost of a given workload. The denominator refers to the number of cycles measured in the execution. This parameter represents the computational density of the algorithm. In the Ideal case with a Single Core this value is 1, with multiple cores is the number of cores.

- CPI (Cycles per Instruction): this metric represents how many cycles are done for each instruction, in average (num Cycles/num Instruction). The theoretical minimum is 1, which means that each instruction is processed in 1 cycle. Some instructions have dependencies, or there are memory accesses, so the pipeline cannot proceed every cycle; that lower the CPI, due to stalls. The closer is the CPI to 1, the faster is the program to run with respect to the number of instructions.

- Latency: this metric is the interval of time that the program needs to run. Usually is derived by the number of cycles with respect to the clock frequency of the processor (cycles/freq). This produces a result in seconds.

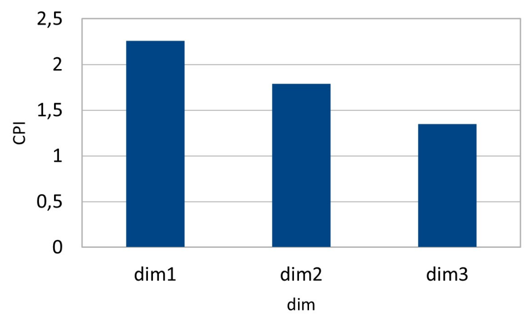
The parameters that describe the net are:

- *dim* that corresponds to the net's dimension. It is chosen 3 combinations of input, output, seq size, presented as dim1 (10, 10, 10), dim2 (50, 10, 50), dim3 (50, 50, 50).
- *core* that represents the number of cores that are used to run, in PULP architecture. (1 to 8)
- *mm_ type* that is the type of mm is used to compute matrix multiplication
- *tanh_ type* that is the type of tanh is used to compute the hyperbolic tangent

To evaluate the impact of mm, tanh, and multicore execution, we present a set of experiments which focus on each of this aspects. The first one presented is the dimension on a single core, then more cores are added, then the different mm algorithms are interchanged, and finally, the different tanh algorithms are compared. It is presented firstly, the Forward algorithm, then, the Backward algorithm, because are done different considerations on them.

4.2 FORWARD

4.2.1 Single Core



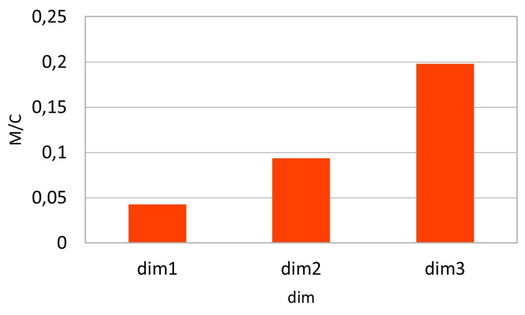
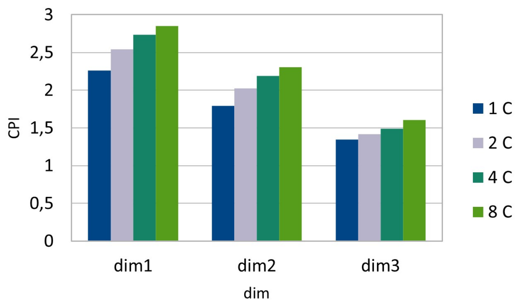


Figure 11: CPI & M/C function of dimension of the net
(1 core $\parallel dim1, 2, 3 \parallel mm \parallel tanh \parallel FW$)

The Fig 11 shows that the M/C increases and CPI decreases with a higher net dimension. A bigger net leads to a bigger matrix size and the operation of matrix multiplication is more efficient in comparison to the other parts of the code. Furthermore, the performance of the mm improves by scaling up the dimensions of the operand matrices.
The CPI of the mm decreases by 24% from dim1 to dim3 up to 1,36.
The M/C of the mm increases by 4,28 times from dim1 to dim3 up to 1,37.
The parts of the code that are less efficient are the initial variables declaration (CPI=3,32) and the mm setup (CPI=4,68). These parts of the code don't scale up with a bigger net and need almost a constant number of cycles.

If the net dimension increases, we get bigger matrices and the part of the cycles due to the only matmul becomes more and more relevant. In dim1 the matmul contributes to 34% of the cycles, and in dim3 mm contributes to 88% of the cycles. Because of that, the cycles of the other parts of the code become more and more negligible, so, the total performance gets closer to the one of the matmul.
The real performance is always lower than the ideal ones since the theoretical calculations neglects numerous overheads, such as instruction cache misses, load and store stalls and the computation of tanh.

4.2.2 Multicore



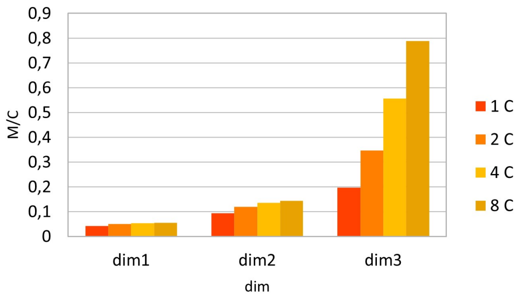


Figure 12: CPI & M/C function of number of cores and dimension of the net
(dim 1,2,3 $\|$ $cores 1, 2, 4, 8$ $\|$ mm $\|$ $tanh$ $\|$ FW)

PULP enables multicore execution, this lower the computational time and improve the values of M/C metric. The workload is distributed on more cores, so the computational time lowers as expected.
The CPI on more cores is slightly higher. Increasing the cores, the instructions are split on all of them and the total cycles decrease as well as the instructions. However the ratio between these two quantities increases, because the number of cycles decreases slower than the number of instructions. This is caused by the parallelization that adds the computational extra effort and delay due to the stalls caused by the synchronization barriers, and TCDM contentions.

Looking at Fig 12, the M/C graph in dim 3, the speedup from 1 to 8 cores reaches 3,9, so, the theoretical maximum Speed Up, considering the whole workload split into 8 parts, is not even approximated. This is caused especially by the tanh algorithm that is sequential, but also by other factors, for example, the current loop chunking.

4.2.3 Matrix Multiplication algorithms

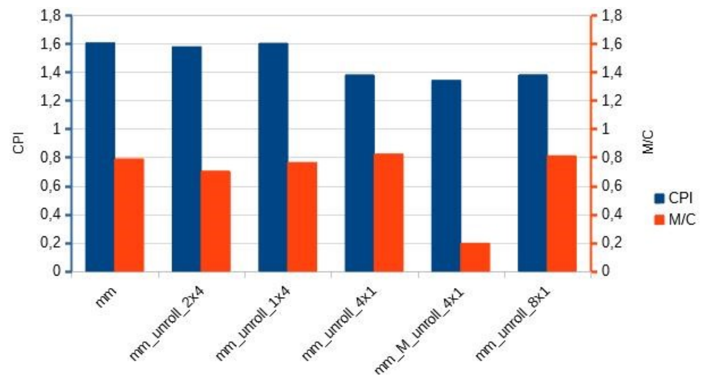


Figure 13: CPI & M/C function of matmul algorithms
(dim 3 ∥ $8cores$ ∥ mm_type ∥ $tanh$ ∥ FW)

In Fig 13 are presented the performances over more types of matmul algorithms. These different results are because of the different unrolling values, U rows, and V columns of the output matrix concurrently computed within the inner loop of the mm. The net has a size of (50, 50), and the most performant algorithm for this dimension is the mm_unroll_4x1.

4.2.4 Tanh algorithms

The tanh algorithm is analyzed after observing that (at most) 77% of the cycles refer to the only computation of tanh. This percentage decreases with bigger dimensions but represents a big overhead. It is clear that performance increases enormously considering the RNN without the tanh, but the results that the model generates are obviously wrong.
Other tanh algorithms are explored and parallelized.

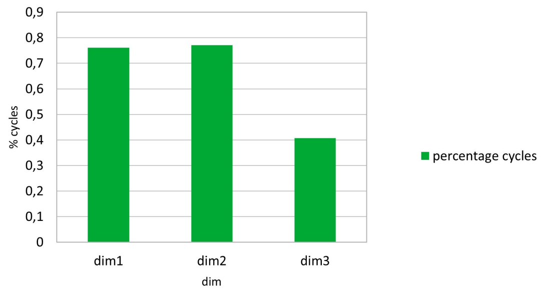


Figure 14: percentage of cycles due to presence of tanh function
(dim 1,2,3 ∥ $8cores$ ∥ mm_unroll_4x1 ∥ $tanh, notanh$ ∥ FW)

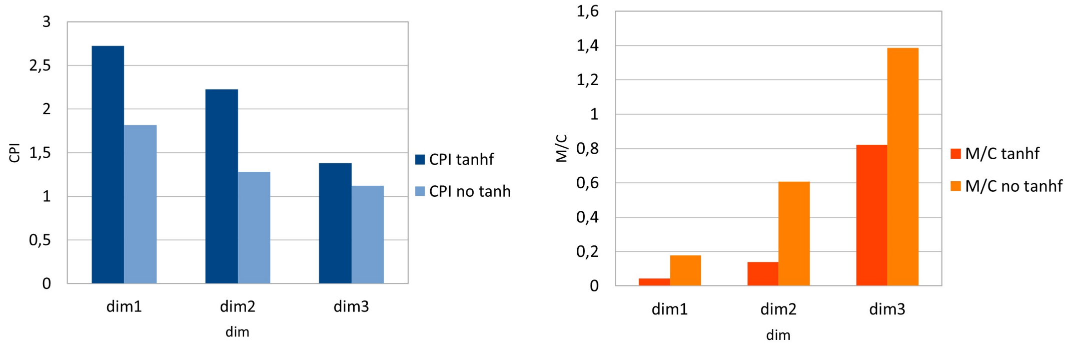


Figure 15: CPI & M/C function of presence of tanh function
(dim 1,2,3 || $8 cores$ || mm_unroll_4x1 || $tanh, notanh$ || FW)

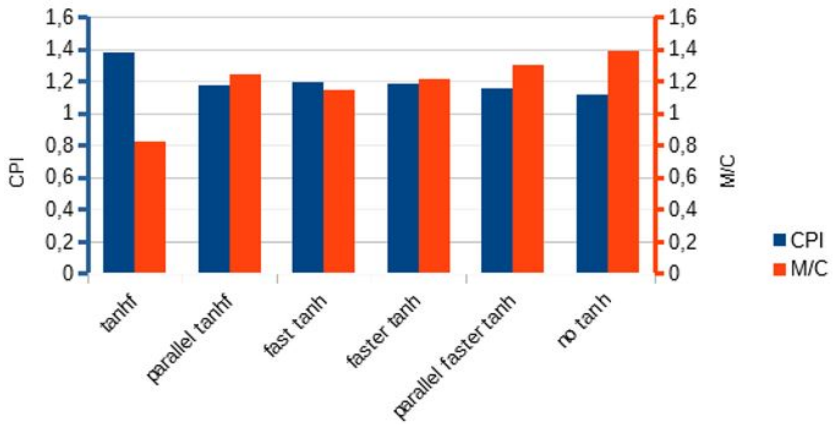


Figure 16: CPI & M/C function of tanh algorithms
(dim3 || $8 core$ || mm_unroll_4x1 || $tanh\_type$ || FW)

In Fig 16 the CPI lower bound and the M/C upper bound of the *no tanh* are the maximum theoretical performance results achievable as they do not include the time to compute the tanh activation, which we consider as an overhead. These upper bounds are far from the results of the *tanhf*, but with other algorithms these values are approached, in particular with the *parallel faster tanh*. It reaches values of CPI and M/C that differentiate to no tanh case of less than 6%.

4.2.5 Parallelization

The theoretical maximum speedup is obtained as a result of Amdahl's law. The chosen dimension is dim3, which consists of Input=50, output=50. So the computation of rows (or columns with the mm_M) is split into different cores. The number of rows and columns, 50, is not a multiple of the number of cores, in the case of 4 and 8 cores. So the workload is split in an unbalanced way between the cores. The result is that the maximum speedup is different to the number of cores added. For 4 Cores, the computed rows per core are:

$$50/4 = 12,5 \;\; \text{split in} \begin{cases} 16 \text{ rows for 3 Cores} \\ 2 \text{ rows for 1 Cores} \end{cases}$$

So the max theoretical speedup for 4 Cores is 50/13=3,8

For 8 Cores, the computed rows per core are:

$$50/8 = 6,25 \;\; \text{split in} \begin{cases} 7 \text{ rows for 7 Cores} \\ 1 \text{ rows for 1 Cores} \end{cases}$$

So the max theoretical speedup for 8 Cores is 50/7=7,1

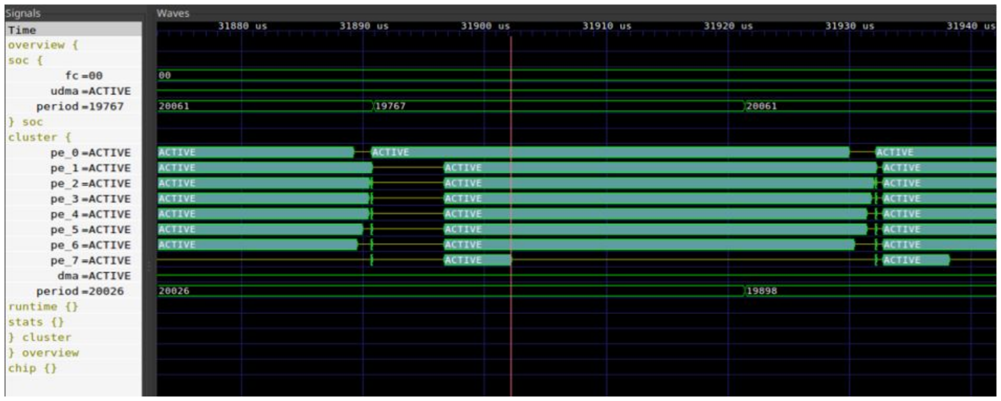


Figure 17: vcd representing the active intervals of the cores
(dim3 $\|$ $8core$ $\|$ mm_unroll_4x1 $\|$ $faster\_tanh\_parallelized$ $\|$ FW)

From the vcd is possible to visualize the state of several components over time, especially the intervals when the cores are active or not. In Fig 16 is represented a parallelization of tanh. It is clear that a first interval is not parallelized, in fact, only the first PE is active. This part is responsible for the sum of the 2 results of the matmuls, the setup of the tanh struct, and consequently, the calling of the fork function.
Then the other 7 cores are activated. The interval of time that cores 1 to 6 are active is 6,4 times the active interval of the 7th core. That means that the 7th core computes a single row and then the remaining 50-1 = 49 are equally split into the other 7 cores, 7 rows each.
This is the workload splitting expected, reported in the formula above.

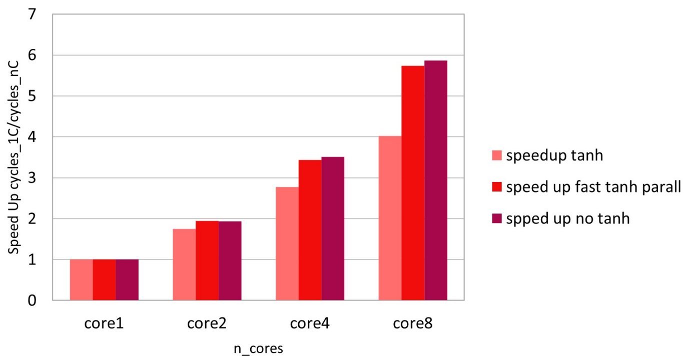


Figure 18: Speed Up on 1C/nC function of cores
(dim 3 $\parallel$ $cores1/2/4/8$ $\parallel$ mm $\parallel$ $tanh, faster, no$ $\parallel$ FW)

The difference between the theoretical calculation and experimental results of speedup in dim3 are:

1 to 2 cores: 2 - 1,94 = 0,06	Er = 3%
1 to 4 cores: 3,8 - 3,43 = 0,37	Er = 9,7%
1 to 8 cores: 7,1 - 5,74 = 1,36	Er = 19,1%

It is notable that the speedup of 2 cores on parallel faster tanh is higher than no tanh one. This highlight the fact that the parallel faster tanh number of cycles decreases faster from 1 to 2 cores, with respect to the no tanh, even if the cycles of the latter one remain lower.

4.2.6 Approximations

In this section are treated some considerations on errors and approximations, because the new tanh algorithms introduced, use a larger approximation range. It is a trade-off, to reach better performances the precision is lower.
Depending on the application, and the impact on the final training accuracy, a threshold can be set, in terms of absolute error and required latency, to define which algorithm fits the application.

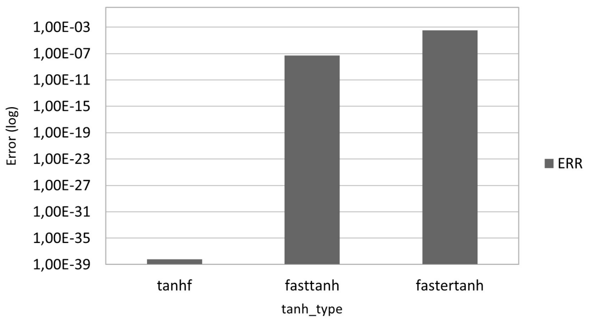


Figure 19: Error tanh algorithms (dim3 $\parallel$ $8core$ $\parallel$ mm_unroll_4x1 $\parallel$ $tanh\_type$ $\parallel$ FW)

With the faster tanh in respect to tanhf the errors increase of $5e+34$ times, reaching an absolute value of $3,4e-4$. But the CPI and M/C of the net improve respectively of 16% and 58%.

4.3 BACKWARD

4.3.1 Single Core

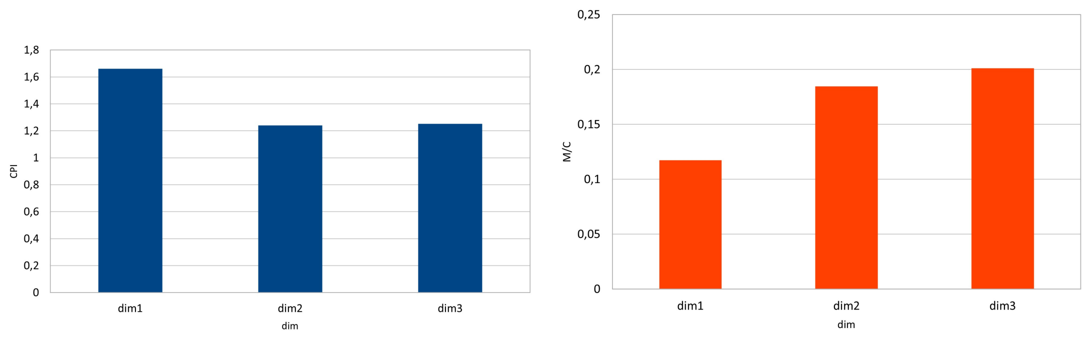


Figure 20: CPI & M/C function of dimension of the net
(1 core $\|$ $dim1, 2, 3$ $\|$ mm $\|$ $tanh$ $\|$ FW)

In this section, we present the results concerning the execution of a Backward step.
The performance increases with a higher net's dimension for the same reasons explained in the Forward section. during backpropagation, the impact of the tanh activation is reduced to a polynomial function. This depends on the backpropagation algorithm, which requires the derivative of the activation function to correctly compute the input gradient of a given layer. So the overhead problem solved in the Forward section, in this part is not relevant.

4.3.2 Multicore

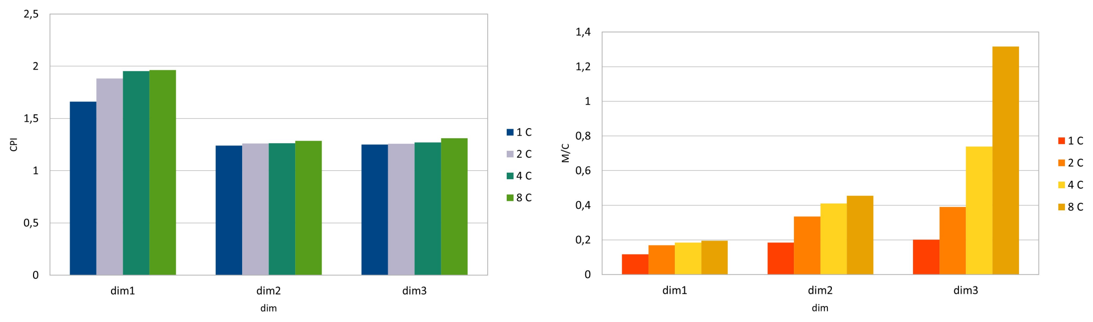


Figure 21: CPI & M/C function of number of cores and dimension
(dim 1,2,3 $\|$ $cores1, 2, 4, 8$ $\|$ mm $\|$ BW)

Fig 21 shows that the CPI slightly increases with more cores, especially in dim1, on higher dimensions is constant. Parallelization has almost no influence on the number of cycles with respect to the number of instructions.
On the other hand, the M/C is heavily dependent on the number of cores. The value of speed up is dependent on the dimensions.
The difference between theoretical calculation and experimental results of Speed up in dim3 are:

1 to 2 cores:	2 - 1,94 = 0,06	Er = 3%
1 to 4 cores:	3,8 - 3,67 = 0,13	Er = 3,4%
1 to 8 cores:	7,1 - 6,54 = 0,56	Er = 7,8%

Furthemore the first two matmulm are done with mm algorithm, altought the third one is computed with the mm$_M$.

4.3.3 Matrix Multiplication algorithms

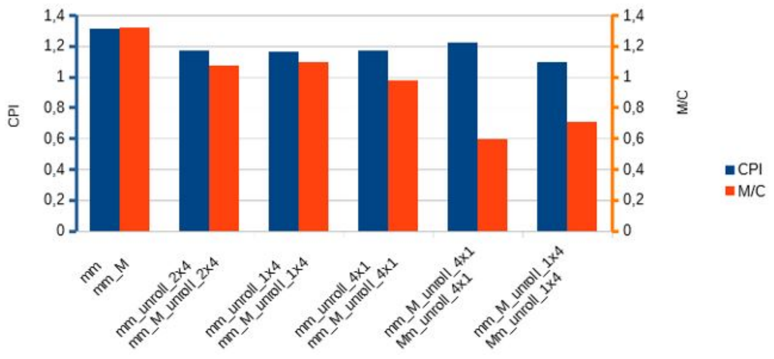


Figure 22: CPI M/C function of MM algorithms
(dim 3 $\|$ $8cores$ $\|$ mm_type $\|$ BW)

In Fig 22 the first matmul function name labeled on the X axis refers to the first and second matmul computed in the Bacward. The second matmul function name labeled refers to the computation of the third matmul.
The performances changing the matmul algorithm are not as similar to each other as in the Forward case. Furthermore, is more efficient to compute the first two matmul with the parallelization on rows (mm), although the third one with the parallelization on columns (mm_M). The reason for this is that the third matmul is a vector-matrix multiplication, so, the result is a vector in which the n-th element is computed with a multiplication of the vector with the n-th column of the matrix. This process is very inefficient to parallelize on the output matrix rows, because the output is a single row, so all the workload is on a single core, and the maximum unbalance is achieved.
On the other side, parallelizing on the output columns means that every N (number of cores) elements of the output vector are computed simultaneously. In absolute terms a speedup of 6,14 it's achieved, with the mm_M in respect to the mm, in the computation of the third matmul. It is a trade off between CPI and M/C, but the best algorithm is the mm.

4.4 Latency

In this section it is summarized and compared the results of the cycles of a single step of Forward and Backward together.

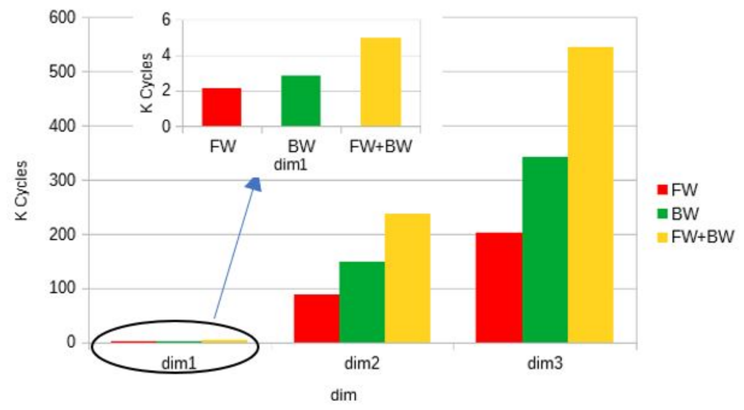


Figure 23: Total number of cycles of FW, BW, FW+BW
(dim 3 || $8cores$ || mm || FW, BW)

The cycles of the Backward algorithm are on average 1,6 times the cycles of the Forward. In fact, the Backward algorithm is computationally heavier than the Forward. Forward comprehend 2 matmuls and the tanh, each recursion.
Backward comprehend 3 matmuls and the update of all the weights' gradients, each recursion.

To exemplify the methodology that has been explained, we provide results evaluated on Vega. This SoC features 10 RISCV cores, comprising a single core for IO management and a 9-Core Cluster, which supports multi-precision SIMD integer (8, 16, 32) and floating-point (FP32, FP16, BFloat16) computation, supported by two Machine Learning accelerators. It also presents 3 levels of memory: 128kB on L1, 1600kB of state-retentive L2 SRAM, and 4MB of non-volatile MRAM. With the aid of these and other advanced hardware features, a power supply of 0.5V to 0.8V, and an operative frequency range of 32kHz up to 450MHz, this SoC is capable of achieving up to 32.2GOPS, with a power envelope scalable from 1:7µW in fully retentive cognitive sleep, up to 49:4mW when at peak perforomance.

Figure 24 shows the latency results to run Forward and Backward propagation steps on Vega, at a frequency of 450 MHz.

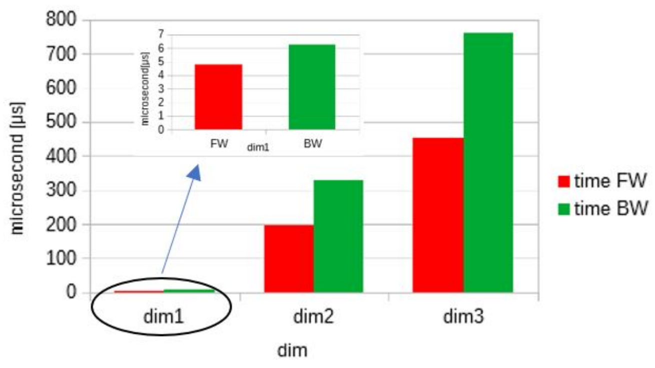


Figure 24: Latency to compute FW, BW in microseconds
(dim 3 || $8cores$ || mm || FW, BW)

4.5 Memory Occupation

In this section, we consider the memory footprint required to perform Forward and Backward steps of a RNN model on PULP. The contributions considered to calculate the memory footprint are:
. Input data + input gradient
. States data + states gradient
. Weight data + weight gradient
Fig 25 presents the estimated memory occupation of this layer.

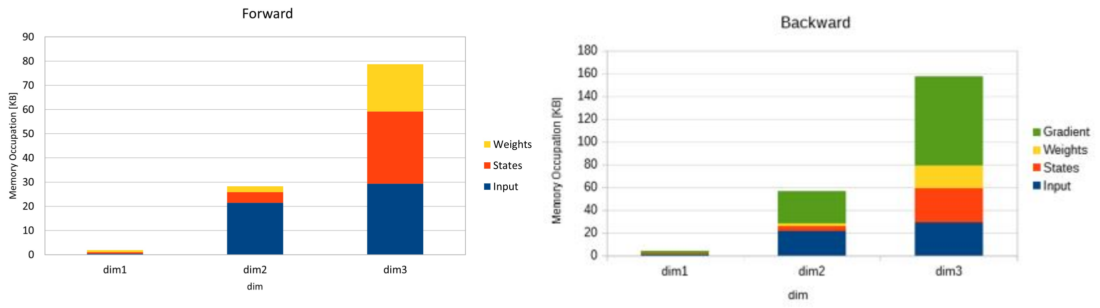


Figure 25: Memory occupation of the net in KByte
(dim 1,2,3 || $8cores$ || mm || FW, BW)

The Forward memory requirment in the biggest tested dimension are 78,7KB, and the Backward 157,4KB.
It is considered that, all of the inputs, states and weights parameters store their gradients as variables, so their memory occupation is doubled in the Backward. The presented memory requirements fit the L1 and L2 memories of PULP-based device like Vega, allowing RNN model training on the IoT end node.

5 Conclusion

It is presented the performance results of a RNN model developed on PULP.
To improve the kernel's performance many strategies have been explored.
Firstly, increasing the dimensionality of the net, secondly, executing on multiple Cores, and finally, many matmul algorithms are tested.
A bottleneck has been recognized in the tanh function, consequently, it has been parallelized and replaced by different algorithms. An analysis of the trade-off performance-approximation introduced has been conducted.
The theoretical speed up are compared to the empiric results obtained, highlighting their validity.
A deep analysis of the workload splitting has been done.
Finally, are reported the performance results imagining to run on Vega SoC, and the Memory requirements.

References

[1] Dzmitry Bahdanau, KyungHyun Cho, Yoshua Bengio “NEURAL MACHINE TRANSLATION BY JOINTLY LEARNING TO ALIGN AND TRANSLATE" *Published as a conference paper at ICLR 2015*

[2] Pushparaja Murugan "Feed Forward and Backward Run in Deep Convolution Neural Network" *School of Mechanical and Aerospace Engineering, Nanyang Technological Univeristy, Singapore 639815*

[3] "How to Implement RNN", URL:"https://peterroelants.github.io/posts/rnn-implementation-part01/ "

[4] “All you need to know about RNNs", URL: "https://towardsdatascience.com/all-you-need-to-know-about-rnns-e514f0b00c7c"

[5] “All of Recurrent Neural Networks", URL:"https://medium.com/@jianqiangma/all-about-recurrent-neural-networks-9e5ae2936f6e"

[6] “Pytorch [Basics] — Intro to RNN", URL:"https://towardsdatascience.com/pytorch-basics-how-to-train-your-neural-net-intro-to-rnn-cb6ebc594677"

[7] “Attn: Illustrated Attention", URL: "https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee3"

[8] Michael A. Nielsen“Neural Networks and Deep Learning", Determination Press 2015 URL: "http://neuralnetworksanddeeplearning.com/"

[9] Davide Nadalini , Manuele Rusci , Giuseppe Tagliavini, Leonardo Ravaglia, Luca Benini , and Francesco Conti "PULP-TrainLib: Enabling On-Device Training for RISC-V Multicore MCUs Through Performance-Driven Autotuning"

[10] "Pulp Hardware Reference Manual"

[11] Giuseppe Tagliavini "Mastering the PULP GCC toolchain"

[12] Andreas Traber, Michael Gautschi, Pasquale Davide Schiavone "RI5CY: User Manual"

[13] Davide Nadalini, Manuele Rusci, Giuseppe Tagliavini, Leonardo Ravaglia, Luca Benini, and Francesco Conti "PULP-TrainLib: Enabling On-Device Training for RISC-V Multi-Core MCUs through Performance-Driven Autotuning"

[14] Liangzhen Lai, Naveen Suda, Vikas Chandra "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs"

[15] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi and Luca Benini "PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors"

[16] Pulp-platform website "https://pulp-platform.org/index.html"

[17] Chi-Feng Wang "The vanishing gradient problem, 2019." URL:"https://towardsdatascience.com/the-vanishing-gradient-problem- 69bf08b15484"

[18] GreenWaves Technologies. Gap8 manual URL:"https://greenwaves- technologies.com/manuals/BUILD/HOME/html/index.html"

[19] Pulp-Trainlib URL:"https://github.com/dnadalini/pulp-trainlib"

[20] Pytorch RNN, URL:"https://pytorch.org/docs/stable/generated/torch.nn.RNN.html"

[21] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini, "PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision". In: *Journal of Signal Processing Systems* 84.3 (2016), pp. 339–354 (cit. on pp. 3, 13, 14).

[22] Manuele Rusci, Leonardo Ravaglia, Alessio Burrello, Francesco Conti, Matteo Spallanzani, Marcello Zanghieri, "LAB01 Embedded Programming on PULP".

[23] Ketan Doshi, "Transformers Explained Visually (Part 3): Multi-head Attention, deep dive". URL: "https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853"

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin "Attention Is All You Need".

[25] Rohan Jagtap "Transformers Explained". URL:"https://towardsdatascience.com/transformers-explained-65454c0f3fa7"

[26] "Building your Recurrent Neural Network - Step by Step". URL:"https://jmyao17.github.io/Machine_Learning/Sequence/RNN-1.html"

[27] Ashray Saini "In-Depth Explanation Of Recurrent Neural Network". URL:"https://www.analyticsvidhya.com/blog/2021/07/in-depth-explanation-of-recurrent-neural-network/"

[28] "RNN, LSTM & GRU" URL: http://dprogrammer.org/rnn-lstm-gru

Afterword

An Engineer is a person who lives in the world. The abstractions, models and approximations he uses are for the world. His inspiration comes from the things around him, they stimulate his curiosity and push him to go deeper into the mechanism.

The Engineers accept the compromise with reality. They use the Scientific Method to deal with the facts of the world, using the models with the right accuracy and dimension in order to be useful in this investigation.
Mathematical laws are intuition to get the concepts of the world and Physical models are used to make things work.

The Scientific Method is the most elaborated, effective, incontestable approach to reality. It is the ability to use contemporary two levels of reasoning; one theoretical, deductive, logical and one toward concrete objects, with empiric information. Overall, in my opinion, it is the most important intellectual achievement that humans reached. Even if not all aspects of our world can be explained by this.

Science is a big castle of cards, pretty stable, but at the same time multiform.
Science passed through many crises, that enforce it, but they were shocking. The incommensurable crisis, make people understand that there is something in the world that cannot be measured, the human intellect cannot comprehend everything. Nicolò Copernico broke the Ptolemaic model and the Anthropocentric conception with it. Quantum mechanics was inconceivable, using a non-deterministic approach to analyze reality, it is a compromise to understand things.
Science is not a linear progression and it's even impossible to measure on a scale because it is not a constant and continuous path. The next big question we ask ourselves could break the balance and the next answer bring a new point of equilibrium.

Scientific progress is marvelous from the inside, but the technology generated with it is a tool in the hands of people. Powerful and rich people can use science for their aims, giving rules and directions to follow. I believe in the independence of science from power and money, even if it must deal with them to go on. Ethics in scientific work is necessary; the research must be driven by the passion of the matter, but cannot be disinterested in the world itself.

At the end of these 3 years of studying, I feel I have an Engineering approach to the problems even if the practical experience were almost none.
The professors who pushed me the most, stimulating me to study and to be curious are many.
Vittorio Martino for his fantasy and entusiasm in solving integrals.
Nicola Semprini for his passion in the first lecture, telling us the story of the second.
Alessio Gagliardi for the inspirational discussions about science.
Luca Benini for welcoming me in the Laboratory and his team with Manuele Rusci, Davide Nadalini and Alberto Dequino.

Finally to my family.