Deep Learning Based CPU Acceleration

R. Luis Jalabert

December 2018

Abstract

The purpose of this project is to explore new ways of accelerating sequential computer code, and finding out if the machine learning techniques available today are able to help us in this task. The core idea is trying to parallelize during run-time (in a way completely transparent to the programmer) the code that's being executed in the CPU, by snooping on the RAM-CPU communication bus, and, in case an artificial neural network considers this code to be parallelizable, it should be then converted into an FPGA module and executed on it instead. In a first approach to the problem, we used a very simple RISC CPU, called LT16x32, and created several Software applications to aid us in this enterprise. First, a parametric assembly code generator named LoopGen, allows us to create many examples of code with different degrees of parallelism. Next, a program called LoopSim generates the neural network's input datasets by compiling the generated code and simulating its execution on the CPU. Finally, an LSTM neural network called LoopOracle is trained with this data to classify the loops. A classification accuracy of 80% was achieved by the neural network, but much work remains to be done in order to improve these results and to answer the tantalizing question that this project poses: can machines do better than humans in parallelizing computer code?

Acknowledgements

I would like to give thanks to Björn Franke for his input and quick answers to e-mails from a total stranger.

I would like to give thanks to Gunnar Tufte for his time and great ideas which gave me direction in a very foggy beginning.

I would like to give very special and warm thanks to Thomas Fehmel, without whose support and help this project wouldn't have advanced an inch.

And last but not least in any way, I would like to thank my tutor, Kjetil Svarstad, for all his time, dedication and for being courageous enough to agree and embark with me in such uncertain and risky sea of ideas.

1 Introduction

The concept of using co-processors as hardware accelerators for mainstream computers dates back to the late 70's, and enjoyed an increased popularity in the subsequent decade. The first version of such co-processors was Intel's 80387: launched in 1980 [1], it was a dedicated floating-point arithmetic integrated circuit (IC) which was fully compliant with IEEE's 754-1985 standard, and was later embedded in Intel's 80486DX. In more recent years, we have seen a dramatic increase in the use of other kinds of hardware as accelerators, such as GPUs, DSPs and FPGAs, which, with the aid of programming languages such as OpenCL or CUDA, can be utilized to off-load some of the most computationally intensive work from the CPU. In addition, the transition to multi-threaded, multi-core processor designs implies that sequential code will no longer achieve the historical performance gains from advances in technology [2] that they have in the past. The "Dark silicon problem" poses new challenges as well as new opportunities to IC designers: System-on-Chip (SoC) designs, containing many different, specialized accelerators can take advantage of this "extra" available silicon, dynamically turning on and off these parts of the chip as needed. All of this points to an apparently unavoidable necessity of changing the sequential-programming-paradigm to a parallel one. However, many factors play against designers when trying to exploit these parallel technologies: Ahmdals law imposes a harsh theoretical upper-limit to the achievable performance speed-up (even when only a small fraction of the code is sequential in nature); the continued use of sequential legacy code that was written years ago before the advent of mainstream parallel computers; and the sharp increase of the difficulty in writing well-optimized parallel code, which requires experienced (and often expensive) programmers to make efficient use of these circuits.

Another idea, which has proven itself very successful in mainstream CPUs (and other kinds of ICs), is the idea of *learning* from previous executions of code. After all, if a computer is repeating the same thing over and over again, wouldn't it be clever to try to use this information when the CPU steps again into the same part of the program? And thus, the idea of space and time locality was used to implement ever-growing cache memories; branch predictors alleviated the burden of jumping back to the same program address; and speculative execution made good use of otherwise idling units.

But what if CPUs were able to learn *more* from a program, what if they could *adapt* themselves to the current program execution? The USA' Defense Advanced Research Projects Agency (DARPA) has recently invested 1.5 billion USD in a project that aims to develop hardware and software that can be reconfigured in real time, based on the kind of data being processed, adapting the computing architecture for the workload in milliseconds [3]. The recent advances in machine learning (and in deep learning in particular) along with its astonishing results, often surpassing human capabilities in a growing spectrum of fields, poses the question of whether this technology could also be used to increase the performance of a CPU. It's certainly not a crazy idea to use neural networks in this way, neither is it new: today's most advanced x86 CPU ar-

chitecture, AMD's "Zen", utilizes a Perceptron-based Artificial Neural Network (ANN) to implement its branch predictor [4].

With all these ideas in mind, this ambitious project proposes a novel architecture that could help solve the aforementioned problems. The concept behind it, is in its essence quite simple: embedding in a single IC a CPU, an ANN, and reconfigurable logic (such as an FPGA). The ANN should then detect serial code (at run-time, by snooping on the CPU-memory bus) that can be accelerated/parallelized by re-programming it (also during run-time) on the FPGA, rather than having idling, highly specialized dedicated IP cores, which is the case in today's SoC architectures. Then, the next time the CPU tries to execute this code, it will be done by the faster, and more efficient FPGA-based accelerator. The proposed architecture is illustrated in figure 1.

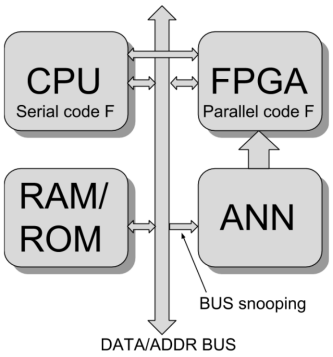


Figure 1: The Basic Blocks of the proposed Hardware

The accelerators running on the FPGA should be dynamically adapted to the code running on the CPU at any given time. Due to limitations in both time and human resources, this project will be limited to generating the training data set and developing an ANN (or at least, investigate its feasibility) that can detect parallelizable code and extract useful features from it. To deal with the extreme complexity of this task, we will only consider loop-level parallelism (as opposed to task-level parallelism), with loops fixed in length and with an upper-bound in the number of instructions per loop. The project will make use of a very simple CPU called LT16x32 which was developed for academic purposes by the Chair of Electronic Design Automation of the Technische Universität Kaiserslautern. This is a uni-core, 16-bit, 3-stage pipeline, 20 instruction, in-order and non-speculative execution CPU. RAM and ROM can be conveniently accessed every clock cycle, therefore avoiding the stalling problems associated with memories running at slower clock speeds than the processor (thus requiring no cache memory). Even though these assumptions can be dangerously unrealistic, all these simplifications are a necessity which will allow us to focus on the core problems at hand, and if successful, future work could improve on the idea by lifting those restrictions. The rest of this document is organized as follows: Section 2.1 describes theoretical background on ANNs and Section 2.2 discusses

topics on loop-level parallelism, while Section 2.3 describes related work. Section from 3 to 5 discusses all the work performed so far. Section 6 describes future work. Finally, Section 7 concludes.

2 Background

This project will focus on the generation of training data sets (instruction-level code) along with the desired training targets (degree of parallelization, number of instructions inside a loop, number of times a loop executes, etc.), as well as on the design and training of an ANN that can detect these parameters.

2.1 Machine Learning and Artificial Neural Networks

ANNs are digital structures loosely inspired on biological brains, consisting on many copies of a basic building block called *neuron*. Their impressive ability to recognize patterns, along with recent breakthroughs in efficient FPGA-based implementations [5] make them an extremely appealing choice for our ideas on runtime loop detection. Even though (to the best of our knowledge) the approach of this project has not yet been attempted, recent work with Recurrent Neural Networks (RNN) in speech recognition, natural language processing, and in particular the results achieved with Long Short Term Memory (LSTM) networks, [6] which seem particularly well-suited for this kind of problem, gives us hope that these techniques will prove themselves effective in attempting to detect desired features by analyzing the sequential opcode patterns generated on the data bus by repeating loops.

A *multilayer perceptron* is a type of ANN which consists of neurons arranged in multiple layers, with the neurons of the l^{th} layer taking their inputs from the previous $(l-1)^{th}$ layer neuron's weighted outputs, and mapped through some *activation function* [7]. Mathematically, in the case of a fully-connected feed-forward network, this output can be written as follows:

$$a_n^l = f\left(\sum_{i=1}^{S^l} w_{n,i}^l a_i^{l-1} + b_n^l\right)$$

where $w_{n,i}^l$ is weight of the i^{th} synapse connected to the input of the n^{th} neuron in the l^{th} layer; b_n^l is a bias term; f is the activation function, and S^l is the number of synapses connected to each neuron in the l^{th} layer. It could be graphically represented as shown in figure 2.

Some common activation functions used in ANNs are the sigmoid function $f(x) = \sigma(x) \equiv \frac{e^x}{e^x+1}$, the hyperbolic tangent function, $f(x) = tanh(x)$ and the rectified linear unit (ReLU), $f(x) = max(0, x)$, being the ReLU a recent and paramount breakthrough on deep learning, since it helps reducing the effects of the *vanishing gradient problem*. Among many other factors, the current success of deep learning can be attributed to some variants of these networks, one of them being the LSTM, which will be described later on.

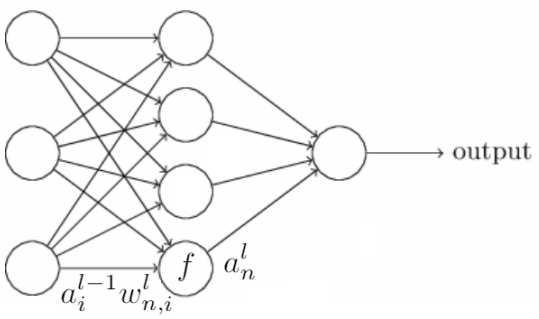


Figure 2: Graphical representation of a multi-layer fully connected ANN

Training of these ANNs require vast amounts of training data and their associated targets (i.e., the desired output for each given input), which the network should learn to predict. This training is done by measuring the error between the network's output and the target, using a *cost function*, and modifying the weights and biases accordingly in order to minimize the cost. The training is generally performed by some form of *Stochastic Gradient Descent* using *Back Propagation* as a gradient computing technique.

2.1.1 Recurrent Neural Networks (RNN) and Long Short Term Memory Cells (LSTM)

In contrast with feed-forward neural networks, RNNs permit connections from a node to itself, which allows them to exhibit temporal dynamic behavior: similar to *Finite State Machines*, they can process temporal sequences of inputs using their internal state. In the diagram shown in figure 3, the neuron gets its inputs from the temporal input sequence x and also from its own state output s. In order to train these networks, we can think of them as multiple copies of the same network, each passing a message from time t to t+1; that is, we can unroll the loop to create a traditional feed-forward network:

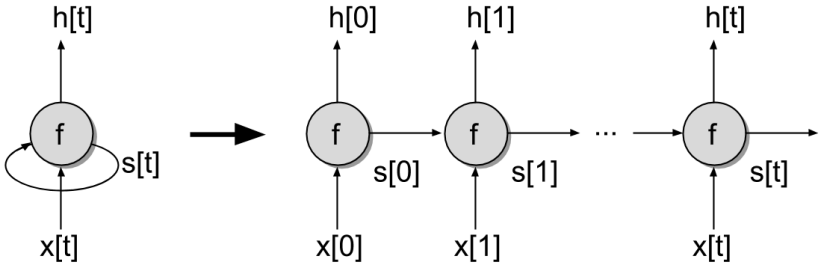


Figure 3: Graphical representation of a recurrent neuron

The LSTM (originally proposed in 1997 by Hochreiter Schmidhuber [8]) is

a special kind of RNN that has the advantage of allowing to connect previous information from an arbitrary point in time, to the present (and thus its name). The internal structure of this kind of cell is depicted in figure 4, and even though it might look complicated at first, it is in its essence, quite simple.

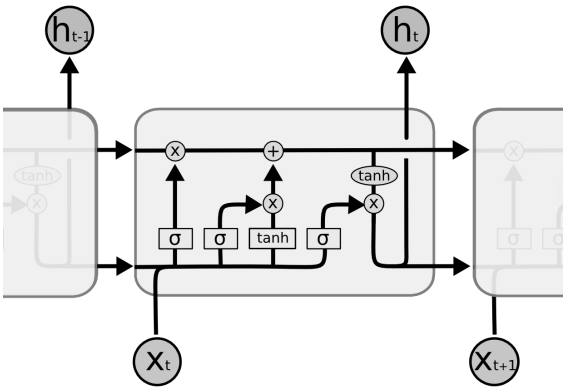


Figure 4: Graphical representation of an LSTM cell, courtesy of Christopher Olah [9]

The core idea of the LSTMs [9] is the *cell state*, the horizontal line running through the top of the diagram. It runs straight down the entire chain, with only some minor alterations. It can add or remove information from its state through structures called *gates*, which consist of a sigmoid layer (that outputs numbers in the interval [0, 1], deciding how much of each component should be allowed to pass), and a multiplier. The first step is to decide what information is going to be thrown away from the cell state. This decision is made by the *forget gate layer*, which multiplies each number in the cell state C_{t-1} by its output f_t, depending on h_{t-1} and x_t : a 1 keeps everything while a 0 discards all of it. Mathematically, it performs the following function:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

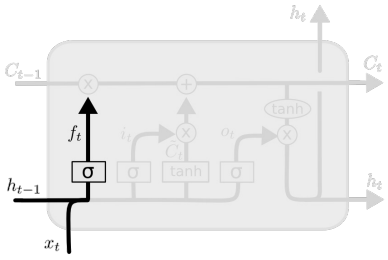


Figure 5: Forget Gate

Next, it decides what new information is going to store in the cell state. This

decision can be divided into two parts: 1) the *input gate layer* decides which values it will update and 2) an *hyperbolic tangent layer* creates a vector of new candidate values $\widetilde{C}_t$ that could be added to the new state C_t. Mathematically, this can be described with the following functions:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\widetilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

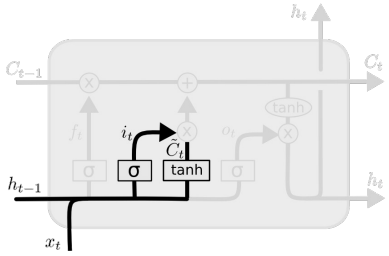


Figure 6: Input Gate

To update the old cell state C_{t-1} into the new cell state C_t, it needs to perform what was decided in the previous steps: the old state is multiplied by f_t and the new candidate values are added, scaled by how much it decided to update each state value. Mathematically:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \widetilde{C}_t$$

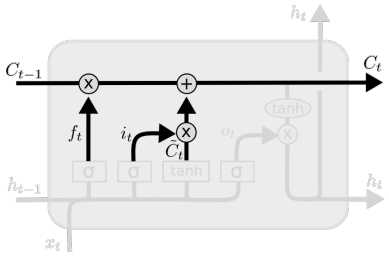


Figure 7: Cell state update

Finally, it decides what its output is going to be. This will be based on a filtered version of the cell state. First, it runs a *sigmoid layer* which decides what parts of the cell state it's going to output, and then it passes the cell state through a *tanh* function and multiplies it by the output of the sigmoid gate, so that it only outputs the parts it decided to. That is:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \cdot tanh(C_t)$$

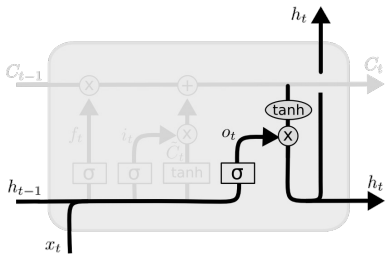


Figure 8: LSTM cell output

LSTMs have performed extremely well in many areas and are being used as fundamental components in commercial products by companies such as Google, Apple, and Microsoft in tasks such as smartphone speech recognition, smart assistants (Amazon's Alexa), Google Translate, etc.

There is currently a substantial amount of research in the area, and there appears to be more promising alternatives such as *attention-based RNNs*, but for this project we will focus on implementing the already established LSTM network, with *fully connected* layers at the input and output.

2.2 Loop-level parallelism and data dependencies

In order to define which high-level characteristics we want the ANN to extract from the code, we must first investigate the *meaning* of loop-level parallelism. As the term itself implies, *Loop-level parallelism* is a form of parallelism that seeks to extract certain tasks from loops which are in general independent from previous executions of the loop body, or in which the data dependencies involved allows the execution of these tasks in parallel or in a pipelined fashion. In case such tasks exist, there are various techniques that can be applied to decrease the execution time, for example: loop-pipelining, loop-unrolling, loop-merging and loop-splitting [10], which are currently being used in high-level-synthesis tools such as Xilinx Vivado.

2.2.1 Data dependencies in sequential statements

To come up with a more precise definition of the *Degree of Parallelism* (DOP) found in a loop, we must previously introduce a few definitions. Given two

sequential statements S_1 and S_2, with $T(S_1) < T(S_2)$, we define the following dependencies[11]:

- True (Flow) Dependence: S1 writes to a location later read from by S2.
- Anti Dependence: S1 reads from a location later written to by S2.
- Output Dependence: S1 and S2 write to the same location.
- Input Dependence: S1 and S2 read from the same location.

To preserve the sequential behaviour of a loop after it has been parallelized, the only kind of dependence that must be preserved is *Flow Dependence*; the others can be dealt with by making copies of the involved variables for each parallel process (this technique is known as *privatization*).

2.2.2 Dependence in loops

When analyzing loops, we can distinguish between two types of dependencies:

- Loop-carried dependence
- Loop-independent dependence

In the case of loop-independent dependence, loops do not present dependence between iterations; therefore each iteration may be performed in parallel. Let's look at an example code for a 2-tap FIR filter:

```
#Iteration independent loop example:
for (int i = 1; i < n; i ++) {
    S1: tmp0 = x[i-1] * tap0;
    S2: tmp1 = x[i] * tap1;
    S3: out[i] = tmp0 + tmp1;
}
```

As we can see, there is a loop-independent, true dependence between (S1,S3) and (S2,S3), but iterations of the loop are independent of each other, therefore allowing for the execution of each iteration in a parallel or distributed manner.

In loop-carried dependence however, statements in an iteration of a loop depend on statements of a previous iteration of the loop.

Let's take a look at an example of code which can be used for calculating the first n values of the Fibonacci sequence:

```
#Loop carried dependence example:
a[0] = 0;
a[1] = 1;
for (int i = 2; i < n; i ++) {
    S1: a[i] = a[i - 1] + a[i - 2];
}
```

In this case, it is clear that the statement S1 depends on the two previous executions of S1, and therefore cannot be distributed across different processing units.

2.2.3 Metrics of parallelism in loops

In order to train our ANN, we need to establish which characteristics or loop parameters we want it to detect. Since we are just beginning to experiment with these ideas, we propose a fairly simple definition of the DOP: *"the degree of parallelism in a loop is the quotient between the number of operations that are iteration-independent and all operations in a basic-block"*. So, in the previous examples, the program "Iteration independent loop example" has a total of three operations, all independent from previous executions of the loop, and therefore a DOP of 1. On the other hand, the program "Loop carried dependence example" has a total of one operation and since it carries a loop dependence, it has a DOP of 0.

It's trivial to see that the following program has a DOP of 0.5:

```
#Mixed loop carried dependence example:
for (int i = 0; i < n; i ++) {
    S1: a[i] = a[i - 1] + b[i] + c[i];
}
```

In this project we will adopt an even simpler view: we will treat loops as either entirely parallelizable (DOP = 1) or entirely non-parallelizable (DOP = 0), which will be given by the *type of loop* (we will clarify this later on), rather than by the dependencies found in the operations themselves.

2.3 Related Work

2.3.1 LoopProf: Dynamic Techniques for Loop Detection and Profiling

As systems transition to multi-core architectures, a greater responsibility is placed on programmers and software for program optimization. According to the work presented in [2], while many tools exist that attempt to automatically parallelize code, manual techniques remain as the most common method of exploiting parallelism. Choosing where to parallelize code is especially difficult for large and complex applications, or for legacy code left by previous programmers. Generally speaking, loops have been the most common targets when trying to find parallelization opportunities. The authors claim that to achieve the best possible speed-up, it's important to focus on coarse-grain parallelization: that is, moving up in the loop hierarchy. This implies the need to have a hierarchical view of the time spent in a certain loop and its nested loops. The tool described in this article, LoopProf, generates a loop call graph and profiles self and total instruction counts, effectively discovering which loops the developer should attempt to parallelize. The paper concludes with a case study evaluating the tool on a variety of SPEC 2000 benchmarks, and warns us that there is still much work to be done in aiding the developer in discovering "how" to parallelize their code.

2.3.2 Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping

The authors of [12] state that compiler-based auto-parallelization hasn't yet been able to find widespread application, largely due to poor identification and exploitation of application parallelism, which results in sub-par performance, far worse than what an expert programmer could potentially achieve. In their research, they have identified two weaknesses in traditional parallelizing compilers and they propose a novel integrated approach which can result in significant performance improvements of the generated parallel code. By utilizing profile-driven parallelism detection (i.e., run-time analysis of the code), they state that the limitations of static analysis can be overcome, which enables the identification of more code parallelism. They replace the traditional target-specific mapping heuristics with a machine-learning-based prediction mechanism (using an SVM), resulting in better mapping decisions (and only rely on the user for final approval) while automating the adaptation to different target architectures. Furthermore, they have evaluated their parallelization strategy on popular benchmarks (such as SPEC 2000) and different multicore platforms. They demonstrate that this approach not only yields significant improvements when compared with state-of-the-art parallelizing compilers but they also come close to (and at times exceed) the performance of code manually parallelized. They achieve 96% (on average) of the performance of the hand-tuned parallel benchmarks on one of the platforms tested, while also achieving significant gains of performance in the other, demonstrating the potential of profile-guided and machine-learning-based parallelization for complex multicore platforms.

2.3.3 FINN: A Framework for Fast, Scalable Binarized Neural Network Inference

In [5], the authors present FINN, a framework for building FPGA accelerators using a heterogeneous streaming architecture. By using a novel set of optimizations, efficient mapping of binarized neural networks (BNNs) to hardware is enabled, allowing the implementation of fully connected, convolutional and pooling layers, with resources being customized to meet throughput requirements. These BNNs are particularly well-suited for FPGA implementations as parameters can be fit entirely in on-chip-memory and enable simplifications in arithmetics, achieving high computational performance. Using a ZC706 FPGA platform, they reach up to 12.3 million image classifications per second with 0.31 μs latency on the MNIST dataset with 95.8% accuracy, while drawing under 25 W, and 21.9 thousand image classifications per second with 283 μs latency on the CIFAR-10 and SVHN datasets with 80.1% and 94.9% accuracy respectively. According to the authors, these are the fastest classification rates reported to date on these benchmarks.

3 LoopGen: A Parametric Assembly Loop Code Generator

As we stated previously, in order to train an ANN capable of recognizing parallelizable loops, we must first have a dataset on which we can train it. Given that the CPU we are working with is an in-house development, there exists no C-compiler for it and we are therefore forced to come up with our own assembly code. Since the number of training examples required is in the order of hundreds, if not thousands, we came up with the idea of generating parametrizable assembly loops, in which the kind of loop, registers used, number and location of memory arrays, number of iterations, etc., are completely parametrizable and so, randomizing these parameters (within certain restrictions), we can come up with hundreds, or even thousands of different assembly programs. Making it possible to change the registers was deemed necessary since we don't want the ANN to be biased by arbitrarily chosen registers which it could learn to associate with certain kinds of loops. This way of creating different assembly programs is, in fact, nothing more than a clever trick to perform *data augmentation* [13]: a machine learning technique which consists in artificially inflating the training set with label-preserving transformations. LoopGen consists of three C++ functions that will be described in the following paragraphs.

The first function, called `"instr_str"`, simply takes the type of instruction and its parameters (such as destination register, source registers, labels, modes, etc.) and returns a string with the correct assembly syntax for the given instruction. This is necessary to make possible the parametrization described before while keeping the registers and labels consistent with the desired behaviour of the assembly program.

For example, calling `instr_str("add", temp_reg0, temp_reg1, temp_reg2, "", "")` will return a string containing `"add r7, r4, r1"`, if the integer values of `temp_reg0, temp_reg1, temp_reg2` were 7, 4 and 1 respectively. Sometimes the same instruction can take different kinds of parameters, which is the case of the branch instruction (it can take an immediate or a register as destination address); other times the same parameter can be specified directly with a number or indirectly with a label, as is the case of the "ldr" instruction, which loads from an address relative to the Program Counter (PC), and so, for example, calling the function `"instr_str("ldr", base_reg0, -1, 0, "", ">ptr_" + arrays_base[0])"` will return a string containing `"ldr r3, >ptr_my_array"`, if `base_reg0 = 3` and `arrays_base[0] = "my_array"`. Here, the -1 in the second position tells the function to use the string `">ptr_my_array"` rather than the second operand as an integer number. More details about this function can be found in its comments.

3.1 Iteration Independent Loops

The four types of loops deemed fully parallelizable will be described below; these loops are returned as a string of LT16 assembly code when calling the following function:

```
loop_independent(int loop_id,const std::string& loop_type, int
temp_reg0, int temp_reg1, int temp_reg2, int temp_reg3, int
n_loops, int n_loops_reg, std::vector<std::string>& arrays_base,
int n_arrays, int base_reg0, int base_reg1, int *fir_taps, int
n_taps, int fir_temp_reg0, int fir_temp_reg1)
```

3.1.1 `"add_constant"` loops

In the case `loop_type == "add_constant"`, this function will generate a program that adds a random value between -128 and 127 (this is limited by the 8-bit immediate taken by the `addi` instruction) to the first `n_loops` elements of the arrays (again limited by the same maximum of 127) given by the vector of strings `arrays_base` and the integer `n_arrays` which indicates the total number of arrays. The code is pretty straightforward, but it's worth mentioning that the amount of instructions depends directly on the number of arrays, so varying `n_arrays` will generate loops of different length (and of course, changing the registers used will also achieve different code). After the main program ends, there is an infinite loop that does nothing; the arrays are defined and memory is allocated for them using assembly directives.

3.1.2 `"add_arrays"` Loops

In the case `loop_type == "add_arrays"` the function will generate code that adds the first `n_loops` elements of the arrays it receives in `arrays_base`, and it will store the result in the first of those arrays. It is in essence very similar to the previous case, with the noteworthy difference that this kind of loop requires at least two arrays in order to work.

3.1.3 `"swap_arrays"` Loops

In the case `loop_type == "swap_arrays"` the function will return code that swaps the first `n_loops` elements of `array_i` with `array_(i+1)` i.e.:
`array_i[j] <= array_(i+1)[j]` and `array_(i+1)[j] <= array_i[j]`. This kind of loop requires an even amount of arrays.

3.1.4 `"FIR_filter"` Loops

By far, the most complex type of loop created was the Finite Impulse Response (FIR) Filter (and its iteration-dependent counterpart, the IIR Filter): creating a parametric, fully functional filter with a CPU that does not possess a multiply instruction, while avoiding crippling its performance by implementing a multiplication subroutine, was not a walk in the park by any means. But I

would argue that whoever doesn't like a good challenge will not have a very bright future in the field of electronics. The implementation is restricted to filtering a single signal given by `arrays_base[0]` whose length is determined by `n_loops`, which was limited to 127 samples; the number of taps were limited to 10 as well. Even though these limitations seem somewhat arbitrary and not strictly necessary, they are required in order to keep the maximum length of the loop body under a reasonable amount (which is around 250 instructions). For a causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values, i.e.:

$$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i],$$

where: $x[n]$ is the input signal, $y[n]$ is the output signal, N is the filter order, b_i is the value of the impulse response at the i^{th} instant.

The ideas implemented here were inspired on the work by Y. C. Lim and B. Liu. in [14], where they demonstrate that an FIR filter with very small frequency response ripple magnitude can be realized using two power-of-two terms for each coefficient value. This allows us to implement a fast filter using shifting instructions as multiplications or divisions. Here we mimic this idea, with the difference that the coefficients were derived simply by calculating the minimum distance from their floating-point representation to a fixed set of values generated by adding and subtracting negative powers of two, i.e.:

$$\arg\min_{p_{1,i}, p_{2,i}} || \pm 2^{-p_{1,i}} \pm 2^{-p_{2,i}} - b_i ||$$

since finding the optimal representation of these coefficients in this set of possible values would be even more outside the scope of this project than this filter implementation already is. Each coefficient is then given by two values passed through the vector `fir_taps`, and they are hard-coded into the immediate values of the `lsr` (Logic Shift Right) opcodes themselves. These instructions take 4-bit immediate values, so the powers would in theory be restricted to the integers in the range [-15, 0]; however, using the number 0 had to be discarded as a possibility. This has a two-fold justification: firstly, the implementation of the `lsr` instruction does not allow an immediate of 0, and secondly, the way we pass the powers to the function is through the integers in `fir_taps`, and the choice of whether a certain power adds or subtracts is simply given by the sign of these integers. Using 0 as a possible power implies that both 2^0 and -2^0 should be possible, but since 0 = -0, the polarity of the coefficient cannot be determined by this method. Fixing this would require an ad-hoc solution, such as passing yet another vector with the signs associated to each power, but since the coefficients of any useful non-amplifying FIR filter are generally less than 1, this was deemed as unnecessary (we will see that this is not the case for the IIR filter, where not even including 0 would solve the issue, but we will present a clever trick to overcome this problem). This gives us the possibility to represent coefficients with modules ranging from $\pm 6.1 \cdot 10^{-5}$ to ± 1.0, which has a certain

reminiscence to floating point number representations, and can be visualized in figure 9.

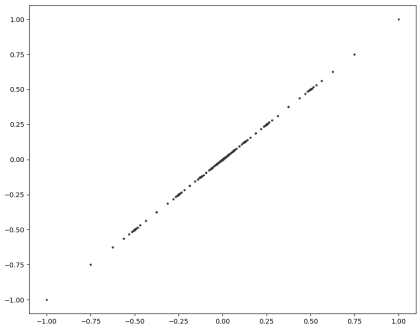


Figure 9: Coefficients achievable by using two negative powers of two

Representing the coefficients in this way produces an error in their representation with an average value of 2.85%, and which never exceeds 10% (except in the vicinity of 0), as can be seen in figure 10. Including a third negative power of two would decrease the average error to a mere 0.49% at the expense of increasing the number of instructions per tap by around 50%.

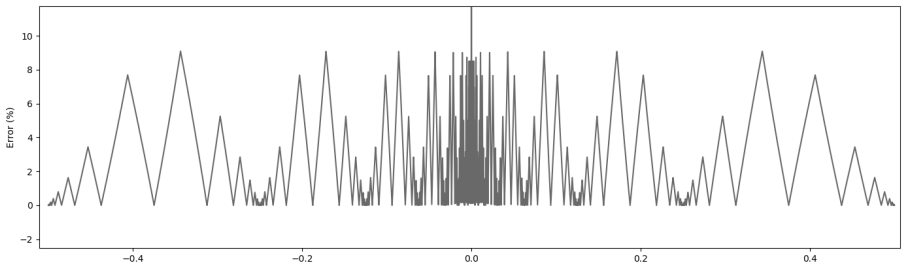
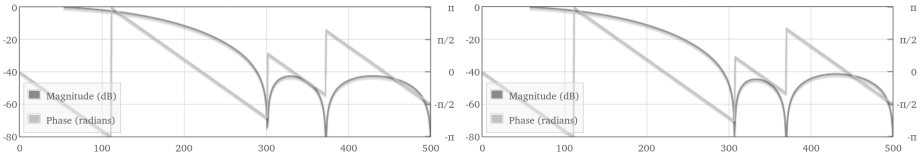


Figure 10: Error in the representation of a floating-point number with two negative powers of two

This of course has a small detrimental effect on the filter's response, which can be observed in the following figures:



(a) Floating-point coefficients

(b) Two powers-of-two coefficients

It can be seen that the example of this particular 10-tap low-pass filter, the attenuation in the stop band is barely altered, improving a little for frequencies

around 300-370 Hz, and degrading for frequencies in the interval 370-500 Hz (using a sampling frequency of 1kHz). All in all, a very good result if we consider that these coefficients are non-optimal.

3.2 Iteration Dependent Loops

The four types of loops deemed non-parallelizable will be described below; these loops are returned as a string of LT16 assembly code when calling the following function:

```
loop_dependent(int loop_id, const std::string& loop_type, int
n_loops, int n_loops_reg, int temp_reg0, int temp_reg1, int
temp_reg2, int temp_reg3, std::vector<std::string>& arrays_base,
int n_arrays, int base_reg0, int base_reg1, int base_reg2, int
*iir_x_taps, int *iir_y_taps, int n_x_taps, int n_y_taps, int
iir_temp_reg0, int iir_temp_reg1)
```

3.2.1 `"fibonacci"` Loops

The first kind of iteration-dependent loop is a function that calculates Fibonacci-like sequences, given by the recurrence formula:

$$f[n] = \sum_{i=1}^{N} f[n-i],$$

with $f[0] = f[1] = ... = f[N] = 1$. It writes the sequence in the array given by `arrays_base[0]`, and when N is equal to 2, it reduces back to the classical Fibonacci sequence.

3.2.2 `"dep_array_sum"` Loops

This kind of loop calculates the sum of elements in an array, for all arrays given in the vector `arrays_base` with each element being added or subtracted depending on whether the previous partial sum was positive or negative. It's useless in real life, but it generates an iteration-dependent loop nonetheless.

3.2.3 `"binom_coeffs"` Loops

The binomial coefficients, which count the number of ways of selecting k elements out of a set of n elements, can be computed using the recurrence:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \leq k \leq n-1$$

with initial/boundary values:

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \text{for all integers } n \geq 0,$$

This can be translated into the following pseudo-code:

```
def binomial(x,y):
    if x == y:
        return 1
    elif y == 0:
        return 1
    else:
        return binomial(x-1,y) + binomial(x-1, y-1)
```

Note that this function will imply loops with variable bounds, and recursive calls to itself. It receives n in `n_x_taps`, and k in `n_y_taps` and returns the calculated value in `temp_reg3`.

3.2.4 `"IIR_filter"` Loops

Much like its finite impulse response counterpart, the Infinite Impulse Response (IIR) Filter was implemented using the same ideas, that is, performing divisions with shift-right instructions. The recursion formula for this type of filter is given by:

$$\sum_{j=0}^{Q} a_j \cdot y[n-i] = \sum_{i=0}^{P} b_i \cdot x[n-i],$$

which can be re-written as:

$$y[n] = \frac{1}{a_0}\left(\sum_{i=0}^{P} b_i \cdot x[n-i] - \sum_{j=1}^{Q} a_j \cdot y[n-j]\right)$$

Unlike in the FIR version, some of the coefficients here are usually greater than two, even when they are normalized by a_0. Since the resolution of the numbers represented with two negative powers of two is quite poor if we don't restrict these values to the interval [-0.5, 0.5], as can clearly be seen in figure 9, the maximum absolute value by which we can multiply is then 0.5, far from the maximum coefficient absolute values we can find in typical IIR filters. One might be tempted to do the following: normalize all the coefficients by 2 times a_0 times their maximum absolute value (i.e., $\widetilde{b_i} = b_i/(2a_0 \cdot max(|a_j|, |b_k|))$, idem for a_i) and with this we would ensure that all coefficients fall in the interval [-0.5, 0.5]. But this would imply reversing these divisions with a multiplication instruction which we do not possess, and representing these multiplications with combinations of positive powers of two achieves very poor representations for numbers greater than 1. Taking one positive power and a negative one carries much the same problem, since all representable numbers would accumulate around the positive powers of two, leaving vast portions of the representable interval empty. However, we don't *have to* normalize the coefficient modules to 0.5; all we have to do is ensure they are *less* than 0.5. This of course can be achieved by dividing by two as many times as necessary (empirically, no more

than two divisions by two were found necessary) in order to keep all coefficients in the desired interval. To revert this division, we can simply use one positive power of two with a shift-left instruction. It could be argued that dividing by two or four could decrease the resolution of the coefficients even further, but this is the case only in the limits where we are using a power of -15 or -14, in which case the associated value would go to zero. But if this is the case, then if the other power is not so small, the relative difference would be very small, and if both powers are really small, then their contribution to the filter is meaningless. There is some speculation in these last words, but further studying the behaviour of these effects on filters is, again, well beyond the scope of this project. The results of applying a low-pass, seven-tap IIR filter implemented with these powers-of-two representations on a noisy sine wave can be seen in figure 12.

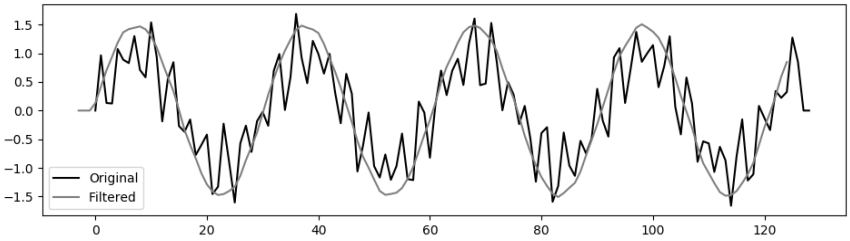


Figure 12: Noisy sine signal before and after applying the assembly IIR filter

One noteworthy problem that is not present in the FIR version is that these filters can easily become unstable with poor representations of their coefficients, since poles and zeros can usually be found near the unity circle, and a small change in these coefficients can turn a stable floating-point represented filter into an unstable one. But for the purposes of this project, the results provided by the filter are meaningless and have no impact whatsoever: all we care about is its recursive, non-parallelizable nature.

4 Generating the datasets with LoopSim: from LoopGen to CPU execution traces

Since we want the ANN to detect loops by inspecting the signals inside a CPU, we must not only create hundreds of assembly programs, but compile them to machine code and produce the traces that would be found in the CPU-memory bus. We must in addition, be able to do all of this in an automated way. Fortunately, the assembly compiler was already implemented, as were some parts of the CPU simulator; solving this was then a matter of adapting the existing code to our needs, and creating what needed creation. The flow of LoopSim can be visualized in figure 13.

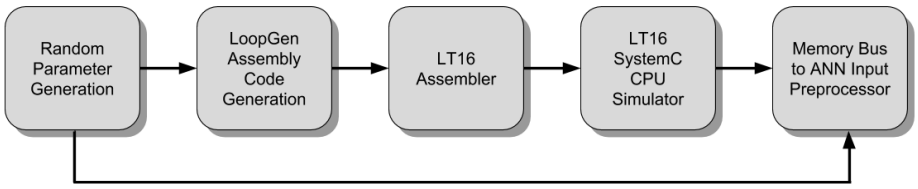


Figure 13: LoopSim Data Flow

First, the parameters to be used with the functions in LoopGen are randomly generated, using a uniform probability distribution, taking care of the restrictions imposed by each kind of loop. The random parameters were carefully chosen in such a way that maximizes the intervals of these uniform distributions, while complying with the restrictions. Some of these parameters also become the targets for the ANN (such as DOP, loop length, etc.). Then, the assembly code generated by LoopGen is assembled into machine code using the compiler provided by the creators of the CPU (which was programmed in C using Flex and Bison), but that had to be embedded in LoopGen and adapted to take C++ string inputs instead of file streams, and also to generate C++ strings as outputs instead of file streams (this might sound trivial but it was in fact, a painful endeavour). A SystemC CPU and its memory models are then instantiated, and the string output from the compiler is then loaded to the RAM model. The CPU model is provided by the so-called "instruction cells", which contains the data types and implementation of the processor and processor state model. Some explanation is required in order to understand how this model works.

Type explanation:

`LT16_v0_1::architecture_state ->` register file of the processor

Special:
`::Active ->` a flag that denotes if that program path is active, for simulation this should always be "1".

`::PC ->` Program counter, i.e., the address of the next instruction to be executed.

`LT16_v0_1::processor_state ->` contains a register file and a memory model.

`LT16_v0_1::processor ->` processor model.
Contains the functions that implement the instructions. Executing an instruction uses the current values of the object.

`LT16_v0_1::processor::S ->` architectural state at the beginning of the in-

struction.

`LT16_v0_1::processor::Z ->` architectural state after the instruction's execution.

`LT16_v0_1::processor::port* ->` are access objects that represent memory accesses to the common memory model.

`LT16_v0_1::processor::port*::Active ->` should be set to '1' prior to accessing 'DATA'.

`LT16_v0_1::processor::port*::Address ->` should be set prior to accessing 'DATA'.

`LT16_v0_1::processor::port*::DATA ->` Proxy object to access the memory model with which the processor was instantiated.

The LT16 has its `PC` in `REGFILE[15]`, but still has `arch_state::PC`, because the description language for the instruction cells require it.

The execution of any instruction is performed by calling the associated function in the processor class, e.g., if we want to add two registers into a third one, we must call the function `LT16_v0_1::processor::ADD(sc_uint<4> Rd, sc_uint<4> Rb, sc_uint<4> Ra, sc_uint<32> currAddr)`. Some of these instructions had to be implemented since they were missing from the model. Given that we were provided only with a SystemC memory model that can hold machine code, and with a SystemC CPU model which only contains the register file and the functions associated with the instructions, it was necessary to implement a *decoder/execution unit* which implements the operation of the CPU. This was done as follows: after the CPU object is instantiated, it's set into the reset state: the register file, including the PC are set to 0. Then, an instruction is fetched from the memory address given by the PC and it is decoded according to its opcode, and the corresponding processor function is called with the corresponding parameters. The function call generates a new status with which the CPU state is updated (in particular, the PC is incremented), and the process begins once again with a new instruction. In the case of a branching instruction, the branch delay slot and branching address are taken care of.

4.1 ANN Input Pre-processor

After having generated the CPU execution traces, it is convenient to represent this information using one-hot encoded vectors as data input for the ANN. Since the opcodes of the CPU instructions have no ordinal relationship, directly feeding the ANN with these 16-bits numbers and allowing the model to assume a natural ordering between instructions may result in poor performance or unex-

pected results [15]. The ANN features a linear encoding layer before the LSTM which not only reduces drastically the high dimensionality of the input space (thus more computationally efficient for large datasets), but it is also able to learn a convenient representation that can find similarities and differences between the instructions. All instructions contain at least one of the following categories, each of which will be encoded using one-hot vectors:

- **Opcode**
- **Mode**
- **Condition**
- **Destination Register**
- **Source Register A**
- **Source Register B**
- **Immediate value**

The first category, i.e., **Opcode** will be encoded as follows:

`(ADD, SUB, AND, OR, XOR, LSH, RSH, ADDI, CMP, LDIMM, LDPTR, STPTR, BIMM, BREG, CIMM, CREG, TRAP, RETI, BRT, TST).`

Therefore, as an example, an "AND" instruction's opcode will be encoded as: (0, 0, 1, 0, ... 0). This vector has 20 dimensions, since all "load from pointer" instructions (LD08, LD16, LD32) are encoded to the same vector. The same goes for the "store pointer instructions".

The **Mode** category is encoded as follows:

`(EQU, NEQ, GR, GROREQ, LESS, LESSOREQ)`

The **Condition** category is encoded as follows:

`(ALWAYS, TRUE)`

All **Source and Destination registers** are encoded in the same way: a one-hot vector with a 1 in the position corresponding to the register number, e.g., register R0 will be encoded as (1, 0, 0, ... 0), and this vector has 16 dimensions.

As for the **Immediate Values**, since they can be 4 or 8 bits long, we will always represent them with a 256-dimension one-hot vector, with a 1 in the position given by the unsigned integer it represents. This is not scalable to architectures with wider instruction sets, so it remains to be seen whether the immediate values can be fed directly without impacting the ANN's ability to learn. All these one-hot vectors are then concatenated and fed to the ANN in the following way:

`(Opcode, Mode, Condition, Destination Register, Source Register A, Source Register B, Immediate value)`

These vectorial time sequences will be created and stored into a *comma separated values* (.csv) file as the CPU simulator executes each instruction, if the `"output_to_csv"` flag is defined. The execution of an IIR filter loop generates a sequence of vectors, which can be visualized in figure 14.

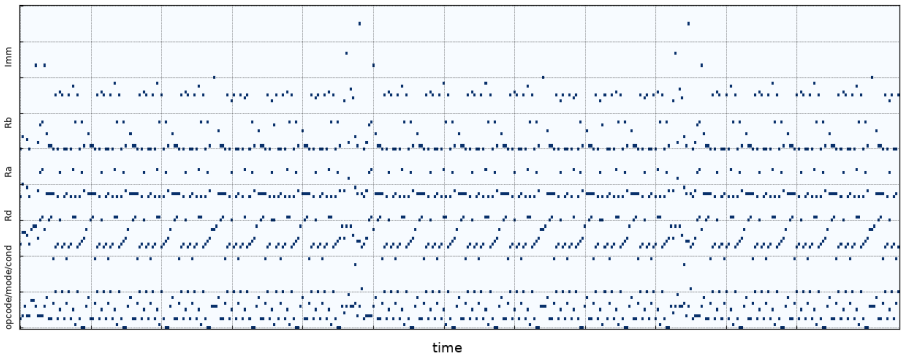


Figure 14: Graphical representation of the first 400 vectorized instructions executed by an IIR filter (only the first 52 dimensions of the 256-dimensional immediate vector are shown)

5 LoopOracle: A Loop Classifier ANN

As it was mentioned previously, the neural network architecture implemented consists of a fully connected layer which takes the vectorized instruction sequences as inputs and creates an adequate encoding for the LSTM network. This input structure, given that it's fed with one-hot vectors, acts as a sort of embedding layer. The LSTM network consists of two stacked LSTM layers with a hidden size of 256. At the output of this layer, a 256-to-2 fully connected layer classifies the loop as either non-parallelizable (output = [0, 1]) or as parallelizable (output = [1, 0]). Graphically, this structure can be visualized in figure 15.

The ANN was implemented using *PyTorch*, an open-source machine learning library for Python (based on Torch) and is used for applications such as natural language processing. It was primarily developed by Facebook's artificial-intelligence research group; it provides a very high-level of abstraction for implementing deep ANNs and therefore it's worthwhile taking a look at the code written, which defines and trains the network. In figure 16, the definition of the ANN class model (which inherits from the class `nn.Module`) can be seen, along its two overridden methods: initialization and forward methods.

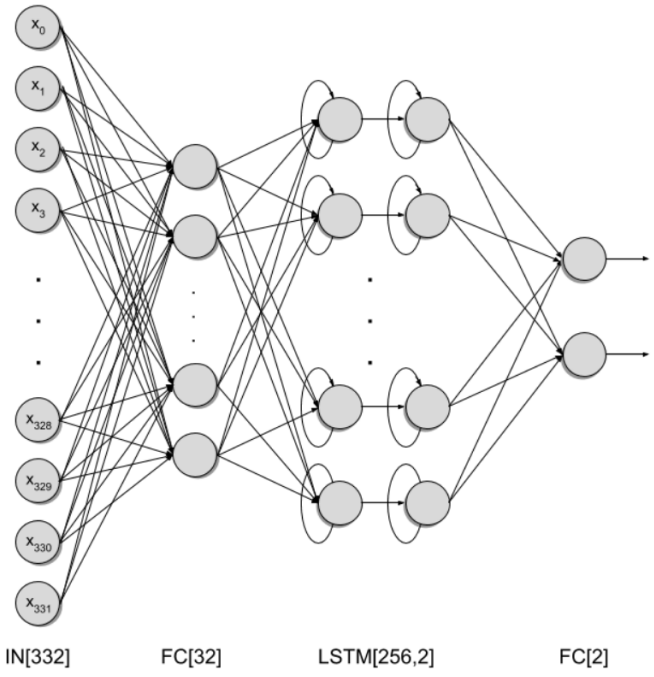


Figure 15: Graphical representation of the neural network structure

```
################
# Define models:
################

class LoopOracle(nn.Module):
    def __init__(self, lstm_hidden_dim, vector_dim, embedding_dim, lstm_layers, output_dim):
        super(LoopOracle, self).__init__()

        # Input layer:
        self.instruction_encoding = nn.Linear(in_features=vector_dim,
                                              out_features=embedding_dim)
        # LSTM layers:
        self.lstm = nn.LSTM(input_size=embedding_dim,
                            hidden_size=lstm_hidden_dim,
                            num_layers=lstm_layers,
                            dropout=0.2)
        # Output layer:
        self.output_layer = nn.Linear(in_features=lstm_hidden_dim,
                                      out_features=output_dim)

    def forward(self, input_sequence, hidden=None):
        output = self.instruction_encoding(input_sequence)
        output, hidden = self.lstm(output, hidden)
        output = self.output_layer(output[-1])
        return output
```

Figure 16: Python code of the model definition

Let's go through the code line by line. The parameters of the ANN can be

easily customized when instantiating a model of the class LoopOracle, which takes as initialization inputs the dimensions of each layer, along the dimensions of the inputs and outputs. The first layer is defined as a linear layer (`nn.Linear`), with input dimensions equal to the number of features in the one-hot input vectors (`vector_dim`), which is currently set to 332, and its output dimensions equal to `embedding_dim`. The next two layers (`nn.LSTM`) are LSTM networks (the number of LSTM layers is given by the parameter `lstm_layers`), which takes the outputs of the previous layer as inputs, and outputs an `lstm_hidden_dim` number of signals. A dropout of 20% means that in every training step, a randomly selected 20% of the nodes in every LSTM layer (except the last), are inactive. This helps the in reducing overfitting by preventing complex co-adaptations on training data [7]. Finally, at the output we have another linear layer (`nn.Linear`) which takes the outputs of the second LSTM layer and performs the classification into two categories (as a first attempt in classifying the loops, we will only consider the two previously mentioned outputs which determine the level of parallelism). So far, we have *defined* the layers but we haven't specified how they are *interconnected*. This is performed in the *forward* method: each time the model is called with an input sequence, it will call the forward function and it will first take this input sequence and forward it through the first fully connected linear layer; this is performed by calling `self.instruction_encoding(input_sequence)`. As we can see, there is no activation function after this layer, since what we want to do is to let the ANN learn an appropriate representation for the input vectors, and this can be done by applying a linear transformation. The outputs of this layer are then passed to the LSTM layer which generates two outputs: the hidden state and the output itself (as it was mentioned in section 2.1.1) . The inputs of the LSTM layer are three-dimensional tensors with dimensions [time x batch size x features]; the outputs are also three-dimensional tensors but with dimensions [time x batch size x hidden dimensions]. Since we only want to take the last time step as an input to the classifier layer, this layer takes as input `output[-1]`, which is the last element of the LSTM's output tensor, and it has dimensions [batch size x hidden dimensions]. This output layer maps the hidden dimensions into the output dimensions.

Now that we have defined the network, it's time to define a training loop: in figure 17 we can see the code that performs this.

Before the definition of the loop, we load the data into an object that inherits from `torch.Dataset`, called `CPUTracesDatasetFromCSV` allowing to easily get items from the dataset. The definition of this object is not shown here, but it provides an iterator, along a length method and `get_batch` method which returns properly sized Torch tensors ready to be used by the model. In the first lines, we can see that the loss function used here was simply the mean squared error (squared L2 norm). The Adam optimization algorithm is an enhanced version of the stochastic gradient descent algorithm, which in contrast to the latter, computes individual adaptive learning rates for the different parameters of the network using the first and second derivatives of the loss function; it also

```
##############
# Train model:
##############

data = CPUTracesDatasetFromCSV("./dataset/")


def train_model(model, learning_rate, num_epochs):
    print("Training started... Please Wait")
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    loss_history = []
    val_history = []
    min_val = 1000
    val_loss_hist = []
    for i in range(num_epochs):
        start = time.time()
        loss_history.append(0)
        val_history.append(0)
        val_loss_hist.append(0)
        # use 80% of dataset as training:
        x_t, y_t = data.get_batch(0, int(0.8*data.__len__()))
        optimizer.zero_grad()
        y_hat = model(x_t)
        loss = criterion(y_hat, y_t)
        loss.backward()
        optimizer.step()
        loss_history[i] += loss.item()

        # Validation:
        with torch.no_grad():
            optimizer.zero_grad()
            # use 15% of dataset as validation:
            x_t, y_t = data.get_batch(int(0.8 * data.__len__()),
                                      int(0.95 * data.__len__()))
            y_hat = model(x_t)
            val_loss = criterion(y_hat, y_t)
            val_loss_hist[i] += val_loss

        if val_loss < min_val:
            min_val = val_loss
            torch.save(model.state_dict(), 'best_model.pt')
        end = time.time()
        print("Epoch: %3d/%3d, Training Loss: %2.5f, Validation Loss: %2.5f, Epoch
              (i+1, num_epochs, loss_history[i], val_loss_hist[i], end - start))

    ###############
    # Show Results:
    ###############

    # Final accuracy test:
    with torch.no_grad():
        # use 5% of dataset as test.
        x_t, y_t = data.get_batch(int(0.95 * data.__len__()), data.__len__())
        y_hat = model(x_t).squeeze(0)
        _, max_ind = torch.max(y_hat, 1)
        y_pred = torch.zeros(y_t.shape)
        for j in range(max_ind .__len__()):
            y_pred[j, max_ind[j]] = 1
        correct = int(torch.sum(y_pred == y_t, dim=0)[0])
        accuracy = 100*correct/y_t.shape[0]
        print("Test accuracy = " + str(accuracy) + "%")
```

Figure 17: Python code of the model training

takes into consideration how fast the weights were changing in the previous steps. The loop itself runs `num_epochs` times, and it uses 80% of the whole dataset for

the training of the ANN. The forward method is called with `model(x_t)` and the predicted results are stored in the variable `y_hat`. After the forward pass, the error (loss) between the target values `y_t` and the predictions is calculated by calling the function `criterion(y_hat, y_t)`. When calling the forward function along the loss calculation, a computational graph of these operations is created, where each operation in this process is registered, allowing to perform the back-propagation pass by simply calling the function `loss.backward()`: this calculates all the gradients of the loss function respect to each parameter in the network. After the gradients are calculated, the Adam optimizer updates the values of the network's weights, and the network is ready to be trained again the next epoch. But before we move on, we first compute the loss in the validation set, which consists of 15% of the whole dataset (however, note that the network does not learn on this data). The idea behind this is having a measure of how well the ANN is performing on new, unseen data, so we can perform *early stopping* [7], which is another regularization technique used to avoid overfitting. For this reason, we keep track of the minimum value of the loss function on the validation set in each iteration, and in case this value decreases from the previous minimum, we store the model into a file that can be loaded in the future. Finally, after training the model `num_epochs` times, we perform a final test using the remaining 5% of the dataset. Here, we calculate the accuracy in the following manner: for each `y_hat` prediction, we equal to 1 the maximum of the two values, and set the other to 0. Then we compare this to the given target values of this particular input, and see if the prediction is correct or not, for all data points in the test set, and then we average these results.

After defining the network and how to train it, we have to actually instantiate the model and call the training function, as can be seen in figure 18.

```
loop_oracle_model = LoopOracle(vector_dim=332,
                               embedding_dim=32,
                               lstm_hidden_dim=256,
                               lstm_layers=2,
                               output_dim=2)


train_model(model=loop_oracle_model,
            learning_rate=0.001,
            num_epochs=250)
```

Figure 18: Python code of the model instantiation and call to the training function

Most of LoopOracle's parameters (in particular the dimensions of the hidden layers and the amount of LSTM layers) can be adjusted in order to increase or decrease the complexity of the network, by simply instantiating it with different numbers of dimensions. However, it must be noted that in order to keep the ANN small enough so that it fits inside an FPGA, these parameters should be kept within reasonable margins. An initial learning rate of 0.001 is a fairly common practise, but we have to keep in mind that when using Adam's algorithm,

this value will be accordingly adapted and it only makes a significant difference in the initial training steps. A number of epochs of 250 is much more than it was found to be necessary, with the validation loss reaching its minimum after around 50 epochs, as can be seen in figure 19. .

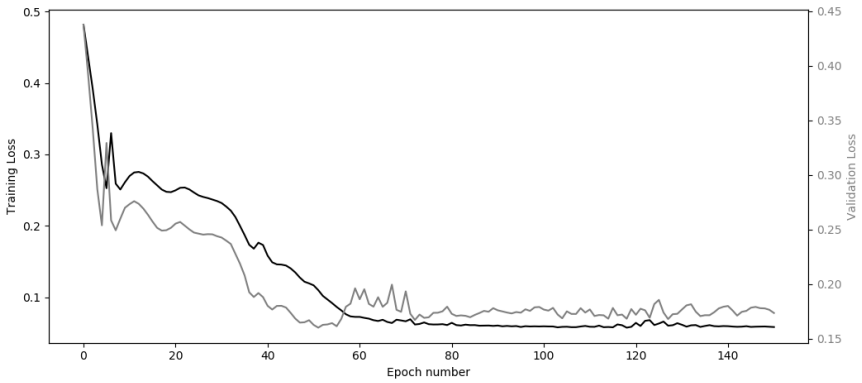


Figure 19: Training and validation loss

A final test accuracy of around 80% was reported.

6 Future Work

As anyone who has read this document so far can guess, this is only a small first step into the idea proposed initially in this project, and therefore, my Master's Thesis will focus on at least two fronts which can be solved in parallel: first, refining and improving the ANN architecture as well as increasing the amount of parameters extracted, and second, implementing the ANN in an FPGA along the bus snooping instruction pre-processor, that vectorizes the instructions to the format required by the ANN. After this, and if enough time is available, the creation of the (or adaptation of an existing) serial-code to parallel-FPGA converter, along the system integration should take place. This would conclude a first prototype of the whole proposed architecture, and should give us a better understanding of the possibilities that it enables. If successful, a future version should adapt the whole design either to an ARM or RISC-V CPU architecture in order to use real-life code, and assess its true performance.

7 Conclusion

Unfortunately (and much to my dismay), the University has not been able to provide me with the hardware required to perform the many tests that are necessary in order to come up with better answers to the questions that this project poses. A request was presented to the High Performance Computing

department for access to computing time, but we haven't received an answer so far. The only hardware that was made available to me (only a couple of weeks before the project's deadline) was a 7-year-old server which is not faster than an average modern laptop. Training this network in my GPU-less computer takes over a whole day using a mere 100-loop dataset with 512 instructions each, making it impossible to test other alternatives that could perform better. Perhaps combining this architecture with convolutional networks could achieve better results, but the limitations of my computer prevents me from increasing its size. Or perhaps including the target address of the load and store instruction as another input to the network could yield better results, since the degree of parallelism highly depends on whether and how these addresses overlap. However, we remain hopeful that in the continuation of this project into my Master's Thesis, the required hardware will be made available, making it possible to test other architectures, possibly answering some of the questions that remain without answer to this day. If we consider that randomly guessing the DOP would achieve an accuracy of 50%, the achieved level of 80% by the proposed architecture is nothing to write home about. It is also not clear from these results whether the ANN is actually detecting features which determine that the code can be parallelized, or if it's simply learning to discern between the types of loops and associating each one with the corresponding output values. The fact that the validation loss does not increase significantly after reaching its minimum, leads me to believe the latter is true. A possible test that could shed some light on the issue would be testing the network with kinds of loops that it hasn't been trained on, but for the reasons stated above (and a lack of time), this was not yet possible. All in all, the progress made in the project seems to be quite substantial, and while it's still unclear whether all this effort will amount to something valuable in practice, it was in my opinion, a quite interesting and challenging academic exercise. It was too a roller-coaster of programming experiences, going back and forth from the lowest level possible (bit-level machine code), through medium level (C and C++) all the way up to the highest level of abstraction that a library such as PyTorch provides. And even though ending up with more questions than answers may leave us with a somewhat bitter taste, it is in my humble opinion, the formulation of the right questions what ultimately drives scientific progress forward.

References

[1] W. Warner. Great moments in microprocessor history, 12 2004. [Online; accessed 14-September-2018].

[2] D. A. Connors R. Ramanujam V. Tovinkere T. Moseley, D. Grunwald and R. Peri. Loopprof: Dynamic techniques for loop detection and profiling. *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications, WBIA '14*, 2006.

[3] HardOCP. Darpa launches 1.5 billion program to reinvent chip technology, July 2018. [Online; accessed 11-September-2018].

[4] Fabien Le Floc'h. The neural network in your cpu, August 2017. [Online; accessed 11-September-2018].

[5] Giulio Gambardella Michaela Blott Philip Leongz Magnus Jahre Yaman Umuroglu, Nicholas J. Fraser and Kees Vissers. Finn: A framework for fast, scalable binarized neural network interface. *FPGA '17 Proceedings of the 2017 ACM/SIGDA International Symposiom on Field- Programmable Gate Arrays*, pages 65–74, 2 2017.

[6] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 5 2015. [Online; accessed 21-October-2018].

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. [Online; accessed 11-September-2018].

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[9] Christopher Olah. Understanding lstms, 8 2015. [Online; accessed 1-December-2018].

[10] Xilinx contributors. Loop pipelining and loop unrolling, 2 2015. [Online; accessed 14-September-2018].

[11] Gina Goff. Practical dependence testing. *PLDI '91 Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 15–29, 6 1991.

[12] Björn Franke Zheng Wang, Georgios Tournavitis and Michael F. P. O'Boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO), Volume 11 Issue 1, February 2014, Article No. 2*, 2014.

[13] Agnieszka Mikołajczyk; Michał Grochowski. Data augmentation for improving deep learning in image classification problem. *2018 International Interdisciplinary PhD Workshop (IIPhDW)*, 2018.

[14] Y. C. Lim and B. Liu. Design of cascade form fir filters with discrete valued coefficients. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(11):1735–1739, Nov 1988.

[15] Jason Brownlee. Why one-hot encode data in machine learning?, 7 2017. [Online; accessed 5-December-2018].