UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RENNÊ SILVA DA SILVA

Using Software Optimization Techniques and Exploiting Hardware Capabilities to Speed-Up BLSTM Neural Network on CPUs

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. Bruno Castro da Silva
Coadvisor: M. Sc. Vladimir Rybalkin

Porto Alegre
July 2017

“There is no path to happiness.
Happiness is the path.”
— Abraham Johannes Muste

ACKNOWLEDGMENTS

First of all, I need to thank my family, especially my mother Ema Rosa, my father Carlos Renato and my sister Renata. They are the strongest pillar of my structure and without them, I would probably never get here. They support me on all my way, giving me everything I've needed and always encouraging me to move forward. The most important thing I've learned from them is education; not a scholar or academic education, but the most important one: moral education. My family always taught me valuable lessons and thanks to them all the care, affection and love they give to me will be passed on to the people who I will meet on my way.

Second, I would like to express my enormous gratitude to M.Sc. Vladimir Rybalkin for guidance, patience and lots of useful insights provided during this work, besides for sharing his vast knowledge. I would like to thank Prof. Dr.-Ing. Norbert Wehn and Taisy Weber by the opportunity of the exchange program in Kaiserslautern, Germany. They allowed me to live one of the most important experiences of all my life. My gratitude to all the staff of the Microelectronic Systems Design Research Group from TU Kaiserslautern for receiving me in an environment where I could feel at home.

I would like to thank all professors of Universidade Federal do Rio Grande do Sul (UFRGS) and structure provided by this university. Besides the UFRGS professors, my gratitude to all my teachers/professors during my way, especially the good ones. They are at some point, part of my personality.

Another important acknowledgment is to all my friends, especially for those who have lived with me for longer periods of time: my colleagues at ADP and colleagues of the exchange program in Germany.

During this year, Prof. Dr. Bruno Castro da Silva helped me a lot as advisor. He really took part in this work. His knowledge, guidance, interest, participation, concern, meaningful insights and support made this graduation work possible and less stressful. My enormous gratitude to him!

Finally, thanks to my girlfriend Bruna Pinzon that with all her dedication, affection and love helped me and encouraged me to move forward.

ABSTRACT

Many current applications benefit from using neural networks to solve machine learning problems, since they often outperform other conventional approaches both in terms of accuracy and robustness. However, training and deploying high accuracy networks sometimes requires using complex neural networks with many weights. This has a direct impact on the time needed to train and run such networks since these procedures involve intensive computations and many memory accesses. Such high processing costs may be a challenging burden even to modern computational platforms such as multi-cores and vector units. Although GPUs can be used in order to improve runtime, they are not always the best option—small networks, for instance, may not benefit from GPUs, and GPUs may not always be available in simpler devices. This graduation work introduces and demonstrates the efficacy of many software optimization techniques that allow for neural networks to fully benefit from the capabilities of CPUs without compromising their accuracy. We evaluate the proposed optimization techniques using a Bidirectional Long Short-Term Memory (BLSTM) neural network to solve an Optical Character Recognition (OCR) problem. Different architectures (Intel and ARM CPUs), memory access patterns, parallelization schemes, linear algebra high-performance libraries, numerical representations, lookup tables and vectorization (SSE, AVX and NEON) were taken into account to specify optimization strategies that allow for improvements in the runtime of the network. Finally, we present performance comparisons between different implementations of a BLSTM, both in terms of runtime and energy consumption, and show that the implemented optimizations improve runtime by a factor of 9 (when compared to an optimized floating-point baseline) while undergoing negligible loss of accuracy.

Keywords: Recurrent Neural Networks. BLSTM. Software Optimization Techniques. Improving Runtime. Energy Consumption Reduction. Parallel and High-Performance Computing. SSE, AVX, NEON Intrinsics.

Usando Técnicas de Otimização de Software e Explorando Capacidades de Hardware para Acelerar Redes Neurais BLSTM em CPUs

RESUMO

Muitas aplicações atuais se beneficiam ao usar redes neurais para solucionar problemas de aprendizado de máquina, visto que elas frequentemente superam outras abordagens convencionais tanto em termos de acurácia e robustez dos resultados. Contudo, algumas vezes, treinar e executar redes com alta acurácia requer o uso de redes neurais complexas com muitos pesos. Isto tem um impacto direto no tempo necessário para treinar e executar uma rede neural, visto que estes procedimentos envolvem computação intensa e muitos acessos à memória. Tais altos custos de processamento podem ser uma carga computacional desafiadora até mesmo para plataformas computacionais modernas tais como as que possuem vários núcleos e unidades vetoriais. Apesar de GPUs poderem ser usadas a fim de melhorar tempo de execução, elas nem sempre são a melhor opção—redes pequenas, por exemplo, podem não se beneficiar do uso de GPUs além de nem sempre elas estarem disponíveis em dispositivos mais simples. Este trabalho de graduação introduz e demonstra a eficácia de várias técnicas de otimização de software que permitem que redes neurais se beneficiem totalmente das capacidades de CPUs sem comprometer sua acurácia. Nós avaliamos as técnicas de otimização propostas ao usar uma rede neural BLSTM para resolver um problema de Reconhecimento Ótico de Caracteres. Arquiteturas diferentes (CPUs Intel e ARM), padrões de acesso a memória, esquemas de paralelização, bibliotecas de alta performance para álgebra linear, representações numéricas, lookup tables e vetorização (SSE, AVX e NEON) foram levadas em consideração neste trabalho para especificar estratégias que permitem melhorias no tempo de execução da rede. Finalmente, nós apresentamos uma comparação entre diferentes implementações de uma BLSTM tanto em termos de tempo de execução quanto de consumo de energia e mostramos que as otimizações implementadas melhoram o tempo de execução por um fator de 9 com perda insignificante de acurácia.

Palavras-chave: Redes Neurais Recorrentes. BLSTM. Técnicas de Otimização de Software. Otimização de Tempo de Execução. Redução de Consumo de Energia. Computação Paralela e de Alta Performance. Funções Intrínsecas SSE, AVX e NEON..

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
GPU	Graphics Processing Unit
ARM	Advanced RISC Machine
LSTM	Long Short-Term Memory
BLSTM	Bidirectional Long Short-Term Memory
NN	Neural Network
ANN	Artificial Neural Network
RNN	Recurrent Neural Network
BRNN	Bidirectional Recurrent Neural Network
CNN	Convolutional Neural Network
OCR	Optical Character Recognition
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Extensions
AVX	Advanced Vector Extensions
NEON	Advanced SIMD Extension for ARM CPUs
BLAS	Basic Linear Algebra Subprogram
ALU	Arithmetic Logic Unit
CTC	Connectionist Temporal Classification
TDP	Thermal Design Power

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Large-scale neural networks, with thousands or millions of weights, are currently being deployed in applications to achieve super-human accuracy. However, training and running such networks is computationally and memory expensive since these procedures involve costly mathematical operations and many memory accesses. Such operations could be run in parallel, in GPUs, thereby achieving a considerable computation speed-up; this is possible since GPUs usually have more resources and faster memory bandwidth, besides providing a better architecture to manipulate vectors. However, affordable and powerful GPUs are not always available for deployment, due to a variety of reasons including cost, component reliability, and programming complexity (VANHOUCKE; SENIOR; MAO, 2011). The overhead of launching GPU kernels and transferring data between the CPU and GPU, for instance, cannot always be amortized over the execution time of some workloads, according to the size of the data (batch size) that must be offloaded. With this in mind, we show that software optimization techniques and hardware capabilities of traditional CPUs can be used and result in significant scalability and performance improvements. Some of the techniques applied in this work are not new to researchers working in high-performance computing, but when they are combined as presented in this work, result in new neural network development methods that provide considerable runtime and energy consumption improvements. We also show that using *specific hardware capabilities* of a target architecture results in expressive runtime improvements even compared to the use of optimized high-performance BLAS libraries.

To develop and evaluate the contributions presented in this work, we will consider the use and optimization of a Bidirectional Long Short Term-Memory (BLSTM) neural network designed for high-accuracy Optical Character Recognition (OCR), when applied to old German text (Fraktur). Optical Character Recognition is the conversion of printed or handwritten text images into machine encoded text. It is a building block of many important visual recognition processes such as data mining, machine translation and text-to-speech translations. BLSTM neural networks are a variant of the Long Short-Term Memory (LSTM) architecture, which is known to often outperform other types of approaches in terms of accuracy and robustness of the results on the task of character recognition (RYBALKIN et al., 2017), (BREUEL et al., 2013), (AFZAL et al., 2015), (GRAVES, 2008), (HAYKIN, 1999) and many other tasks.

BLSTM neural networks are Recurrent Neural Networks (RNNs) that are known

to perform well when processing data with a sequential or temporal characteristic, such as sequences of letters in an image for OCR and sequences of wave forms for speech recognition, where the identification of a given word is made easier by taking into account the words and sounds that preceded it. BLSTMs achieve this capability by implementing a memory-like functionality, capable of preserving information from previous outputs of an input signal to the network as part of an internal state. In the context of OCR applications, BLSTMs are an appropriate machine learning algorithm since they allow for the network to consider both preceding and following characters in an image, when performing the transcription of a specific part of a text.

There are many challenges involved in developing an efficient software implementation of a BLSTM. A first challenge is the high memory bandwidth needed to transfer network weights from memory to computational units (e.g. ALU). Some characteristics of BLSTMs make it difficult to efficiently implement them in terms of better memory usage: first, they have recurrent connections, which makes it harder to parallelize the computation of their outputs; secondly, BLSTMs differ from RNNs in that they have four additional gates to control the data flow within a neuron, which has an impact in the amount of data that needs to be loaded and processed by each individual neuron; thirdly, BLSTMs, unlike classic neural network architectures, read input data in two directions: forward and backward. This means that the hidden layer of the network needs to be duplicated, which has a direct impact on the amount of memory needed to compute the output of the network. In this work we propose to deploy a series of optimization techniques over a baseline BLSTM implementation in order to overcome these difficulties; the techniques are: *1)* rearrange weighs in memory and use new access patterns that allow us to avoid a duplication of the hidden layer; *2)* rearrange weights in memory to have full data vectors in contiguous memory, making better use of cache locality; *3)* reduce the memory required for intermediate results, which can be high due to the fact that BLSTMs have recurrent connections; and *4)* tolerate a small reduction in the accuracy of the network by reducing the numerical precision of the weights that compose it.

A second challenge in efficiently implementing BLSTMs is the high complexity of each LSTM cell structure, which includes activation functions and multiple multiplicative units making use, in particular, of expensive dot product algebraic operations. In order to speed up the baseline implementation of BLSTM that we consider in this work, we suggest to explore runtime improvements by *1)* making use of sequential data vectors for processing high-dimensional inputs; *2)* performing loop unrolling of dot product

operations; *3)* using BLAS library for more efficient dot product operations. Although these standard optimization techniques allow us to accelerate the relevant algebra operations and result in better performance, we aim to further accelerate the implementation of dot products by modifying a baseline BLSTM implementation by also exploiting specific hardware capabilities. We propose, for instance, to do so by allowing more data to be processed by a same instruction, using: *4)* SSE and AVX intrinsic functions (for Intel architectures); and *5)* NEON intrinsic functions (for ARM architectures). And finally, we intend to accelerate BLSTMs by *6)* applying lookup tables to the neuron activation functions, in order to replace runtime-intensive mathematical functions with simpler arrays indexing operations while tolerating a small decrease in accuracy.

To parallelize the neural network, we take into consideration two common scenarios of visual recognition mobile applications: one where the tasks are offloaded to the cloud to be run in high-performance machines, and another where the tasks run locally on an embedded processor. The second scenario has an important concern: energy consumption. In order to reduce this requirement in mobile applications, we also take into consideration the need to reduce the amount of allocated resources. With these two aspects in mind we propose two different parallelization scenarios: one that is better suited for batch processing (where all inputs are loaded in memory), which we call *Input-level parallelization* or *Offline*, and another one that is better suited to process one high-dimensional input or a sequence of inputs at a time, but using fewer CPU resources; we call this the *Neuron-level parallelization* or *Online* scenario.

As previously mentioned, we demonstrate the performance improvements obtained by our proposed techniques in an image recognition application. In our experiments, we use a network composed of one hidden layer with 200 neurons, divided equally into forward and backward direction functions. The test set of our application is composed of 3401 text images. We evaluate the accuracy of the outputs computed by the network, as well as its runtime. The reference recognition accuracy of the baseline BLSTM implementation is 98.2337% using single-precision floating-point format. We consider applying optimizations over two Intel architectures: the Intel I7-4500U, a power-optimized CPU designed for laptops, and the Intel Xeon E5-2670 v3, a high-performance CPU. We also consider two ARM low-power CPUs for embedded devices: the ARM Cortex-A53 (e.g. used by Zynq FPGA board) and the ARM Cortex-A9 (e.g. used by Raspberry pi 3 board). We show that by using our proposed optimizations, it is possible to produce a BLSTM that is 9x faster than a baseline implementation with parallelization. We also

show a significant decrease in energy consumption, while undergoing a negligible loss in accuracy.

1.1 General Objective

In this work, we present contributions to improve performance and energy consumption of neural network software implementations in CPUs without compromising their accuracy. We aim to achieve this objective by applying software optimization techniques, and using hardware capabilities of modern computational platforms, to speed-up the most intensive calculation parts of the code of a BLSTM neural network for Optical Character Recognition.

1.2 Outline

The thesis is structured as follows. In Section 2 we review the theoretical basis that underlies this work and present the different architectures used to evaluate our implementations. In Section 3, we review previous publications that also aim at speeding up neural networks. The software optimization techniques and hardware capabilities explored in this work are described in Section 4. In Section 5, we show the way that we intend to conduct the experimental analysis. Finally, Section 6 presents the results obtained via the proposed techniques and compares different configurations of the implementations. Section 7 concludes this work by outlining its main results and discussing future work.

2 THEORETICAL BACKGROUND

In this chapter we will review the main neural networks concepts that are relevant to the implementation of BLSTM neural networks. We will also introduce the Optical Character Recognition problem, used in this work to evaluate the performance of our optimized neural network, as well and a string comparison algorithm that is used to measure the accuracy of our neural networks. Finally, we present the main computer architectures that are relevant to this work, giving special attention to some particular features of each one of them.

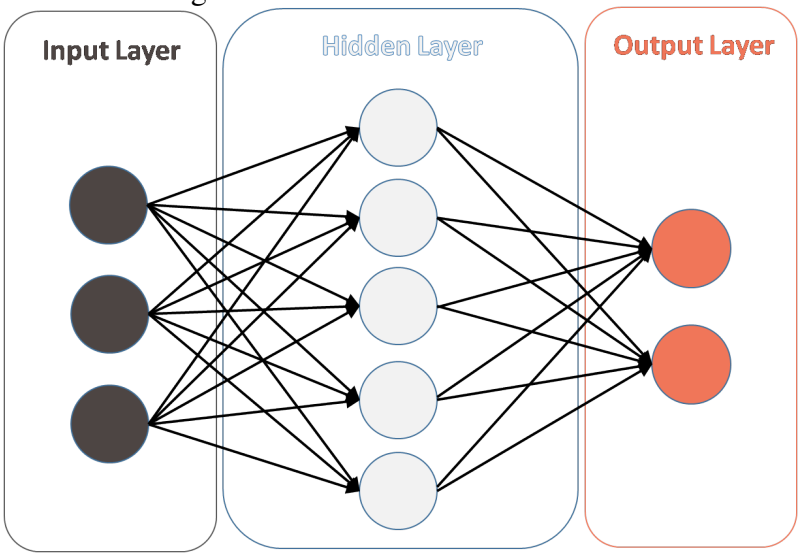
2.1 Neural Networks

Artificial Neural Networks (ANNs or just NNs) are interconnected groups of nodes that computationally model the functionality of the human brain. The goal of NNs is to solve problems in the same way humans would do. Each node of an NN can be identified as a neural unit, and they are connected with many others within a same layer or among layers (see Figure 2.1). The input signals presented to a network travel from the first (input) to the last (output) layer, passing through intermediate layers—either just one (single-layer) or more hidden layers (multi-layer). The activation function of a neuron determine its output, given inputs; they typically compute this by taking a weighted sum of the inputs (weighted by their respective weights) and limiting it to a pre-defined range, before propagating this output to other neurons. Typical activation functions used in neurons include the functions $1/(1+\exp^{-x})$ and $\tanh(x)$. The connections/weights between neurons can enforce or inhibit the effect on the activation state of connected neural units.

NNs need to be trained from examples, rather than explicitly programmed. The training process aims at finding weights that result in the NN correctly mapping given inputs to the most appropriated outputs. There are many algorithms to train a NN and estimate the optimal weights of each neuron. One of the most popular algorithms is called Backpropagation.

The Backpropagation algorithm ((BRYSON; DENHAM; DREYFUS, 1963)) consists in measuring the error of the network when applied to a given input (specifically, by comparing its output with the desired/ground truth output), and then using this information to modify the network weights in order to minimize the error. The Backpropagation algorithm is a supervised learning technique that receives as input a training set (con-

Figure 2.1: Artificial Neural Network



Representation of a feed forward artificial neural network with ten neurons. Each neuron is represented by a circular node. The arrows represent the connections/weights between neurons. In this example, the neurons are split into three layers. Three neurons compose the input layer, five neurons the hidden layer and two neurons compose the output layer.

sisting of sample pairs of input and corresponding desired output) and iterates over its elements in order to optimize the weights of the neurons. At first, all weights are initialized randomly. Following, input samples are propagated forward through the network and its output is compared with the expected one (i.e. ground truth) using a given loss function; if the difference/error is not satisfactory, the error is propagated back to the previous layers of the network and used to adjust their internal weights. After that, another iteration starts, where the next example is presented and weights are adjusted, and the process is repeated until a stop criterion is met.

A simplified version of the method used to determine the output of a neuron is presented in Equation 2.1. The weighted sum of the inputs is represented here by the dot product between an input vector X and the weight vector W of the neuron. This sum is processed by a σ activation function resulting in h, the neuron output (MCCULLOCH; PITTS, 1943):

$$h = \sigma(W \cdot X). \tag{2.1}$$

Among the many different existing ANNs architectures we can highlight those purely built with acyclic connections, also known as feed-forward neural networks (see Figure 2.1), and those with cyclic connections, referred to as Recurrent Neural Networks (RNNs).

2.1.1 Recurrent Neural Networks

RNNs are able to look at recent information (i.e. previous outputs) to perform the present task. The cyclic connections of an RNN architecture allow for a memory-like functionality, preserving the previous activation outputs of the network as a type of internal state (see Figure 2.2). This is helpful for sequence processing tasks that need access to past network outputs in order to improve accuracy at a current input.

Figure 2.2: A recurrent neural network and its unrolling over time

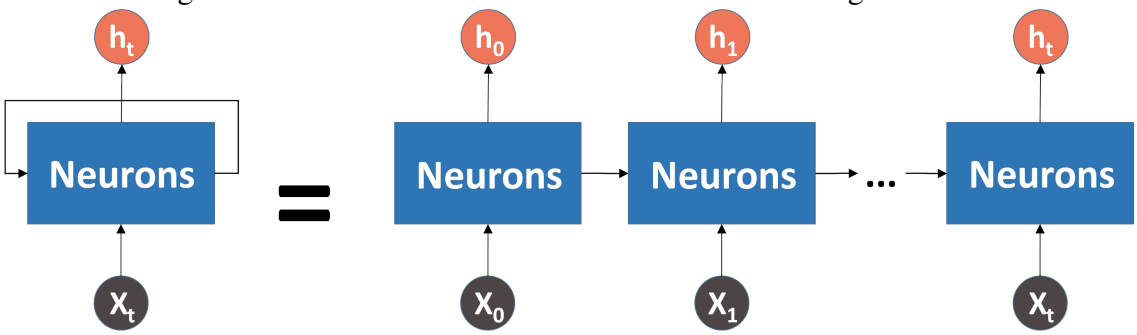


Figure adapted from: (OLAH, 2015). On the left of the equality, an RNN with its cyclic connection. It receives a given input X_t and outputs a value h_t at time t. The loop/cyclic connection allows for information about previous activations to be taken into account at future timesteps, when processing subsequent inputs. On the right of the equality, we see the same RNN but with its loop unrolled in time. The network can be thought of as multiple copies of a same network, repeated over time, each one passing its output as a message to a successor network. The loop unrolled RNN makes it explicit how this architecture keeps track of input sequences, by clarifying how context information about previous inputs/outputs is passed ahead when processing subsequent inputs.

Bidirectional Recurrent Neural Networks (BRNNs) were proposed to take into account previous outputs from both sides of an input signal (e.g. past and future letters of an input image). The input signal is processed both in the forward and backward directions by passing it through two separate hidden layers. The outputs of these layers are added by a common output layer that has access to past and future context of the given input.

Despite being extremely important, RNNs present a problem of long term dependencies. Olah (2015) explains this problem by considering the difficulties of building a language model capable of correctly predicting which words may follow other words:

> *"If we are trying to predict the last word in "the clouds are in the sky", we do not need any further context—it is pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and*

> *the place that it is needed is small, RNNs can learn to use the past information (...) There are cases where we need more context. Consider trying to predict the last word in the text "I grew up in France... I speak fluent French". Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It is entirely possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information"* (OLAH, 2015)

Due to this limited range of accessible context (i.e. the network can keep information just from the recent outputs and not of long-term dependencies) with RNNs, Long Short-Term Memory (LSTM) architecture was proposed in (HOCHREITER; SCHMIDHUBER, 1997) to overcome this problem.

2.1.2 LSTM and BLSTM Neural Networks

LSTM networks are a special type of RNN architecture, capable of storing and encoding long-term dependencies between its inputs. The LSTM architecture replaces the simple nodes (i.e. neurons) of an RNN (such as a *hyperbolic tangent* function) with memory blocks, or "memory cells". Each memory cell (i.e. neuron) has three different internal *gates* that control the flow of the input data and determine in which way the internal state (or memory) of the cell will be updated, as it processes new input signals. The LSTM cells that compose the overall LSTM neural network can be thought as repetitive connections of one LSTM cell with the next one, similarly to the repetitive connections previously shown in Figure 2.2.

The three control gates that compose an LSTM cell are called *forget, input, and output gates*: (f, i and o), respectively. These gating functions can be interpreted as reset, write and read operations, respectively. They are applied to the cell internal state (C) and are responsible for preserving the internal state of the memory cell over longer periods of time. The cell state signal C is propagated through the entire chain of memory cells, undergoing changes and updates determined by the above-mentioned gates. Each gate regulates the way in which state information must be removed or added to the cell state. Each control gate is implemented as a $sigmoid$ neural net layer and a pointwise

multiplication operation; the outputs of this layer are numbers between zero and one, specifying how much of each component of the memory/state should be preserved.

In the case of a *forget gate*, an output of zero can be interpreted as a signal ordering the network to remove information from the current memory state. In an OCR application, for instance, this might happen when we reach the end of a sentence, in which case we can forget the last word that was processed since it may be irrelevant, e.g., to predict if the next word will be a verb or a noun. An output of one, on the other hand, can be interpreted as a signal ordering the network to completely preserve (not forget) a given part of the state, since it may be very relevant to determine the class of upcoming words. A similar intuition applies also to *input gates* (which determine how strongly information about new words should be included in the current state/memory), and to *output gates* (which determines how strongly the current state/memory values will be used in determining the present output or prediction being made by the network). In Figure 2.3, the output of the forget, input, and output gates are depicted, respectively, as the control gates of the cell state.

A complete description of an LSTM cell structure is presented in Figure 2.3. Equation 2.2 presents formulas that compute the forward activation (or output) of an LSTM network (GREFF et al., 2015). In Equation 2.2, t denotes the current time, $t-1$ refers to the previous timestep/input, (X) is an input vector; and (h) refers to the cell's output. These equations specify, in particular, *1)* how the current state/memory, C_t, is updated based on the previous state, C_{t-1} and on the outputs of the control gates; and *2)* how its outputs are updated, h_t, given the current input, X_t. In these equations, (b) is a bias vector, (W) is a rectangular input weight matrix for each gate, and (R) is a square recurrent weight matrix. The so-called *peephole connections* (p) are responsible for allowing the gates to access the cell's internal state. Since the goal of this work is not to provide an in-depth derivation for these equations, we refer the interested reader to (GREFF et al., 2015) and (OLAH, 2015) for more technical details.

$$
\begin{aligned}
f_t &= \sigma(W_f X_t + R_f h_{t-1} + p_f \odot C_{t-1} + b_f) && \text{forget gate} \\
i_t &= \sigma(W_i X_t + R_i h_{t-1} + p_i \odot C_{t-1} + b_i) && \text{input gate} \\
z_t &= \tanh(W_z X_t + R_z h_{t-1} + b_z) && \text{block input} \\
o_t &= \sigma(W_o X_t + R_o h_{t-1} + p_o \odot C_t + b_o) && \text{output gate} \\
C_t &= i_t \odot z_t + f_t \odot C_{t-1} && \text{cell state} \\
h_t &= o_t \odot \tanh(C_t) && \text{block output.}
\end{aligned}
\tag{2.2}
$$

Figure 2.3: LSTM Cell Structure

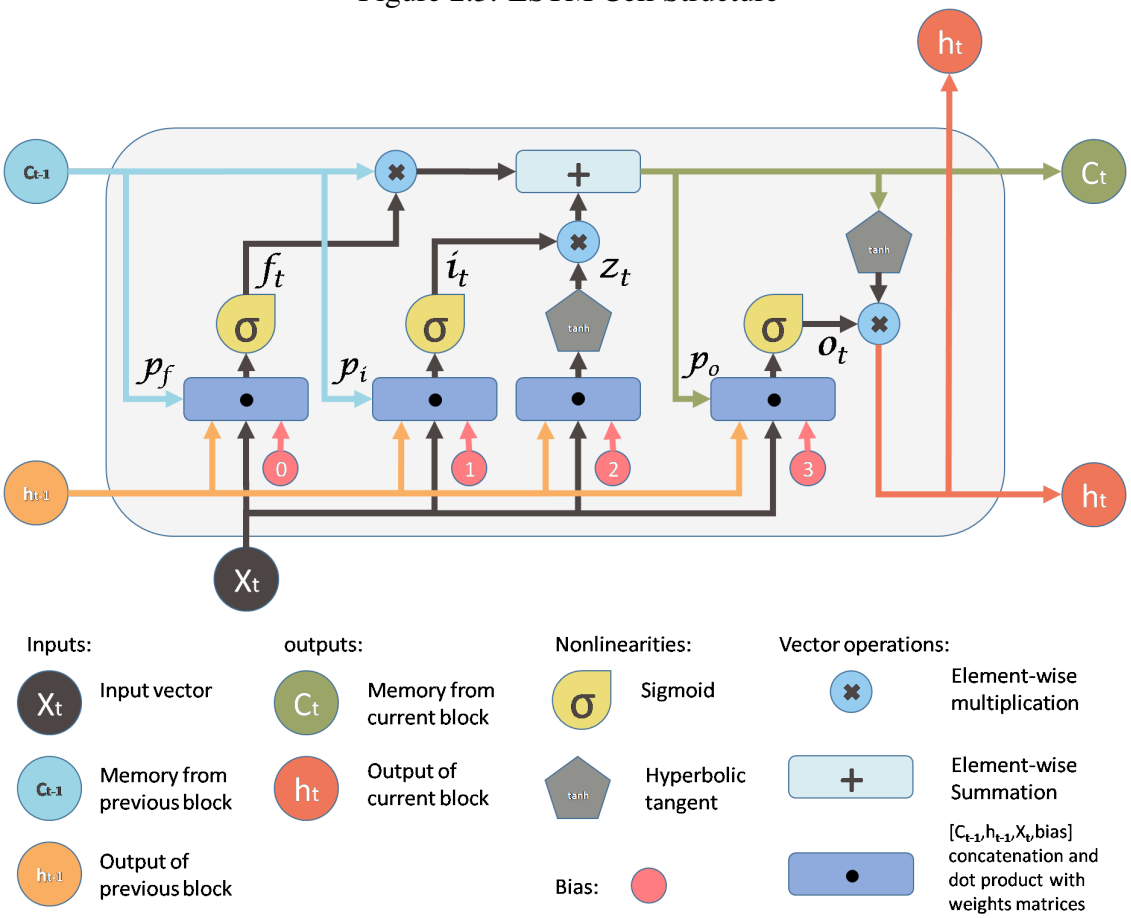


Figure adapted from: (YAN, 2016)

In Figure 2.3, the output of each control gate is processed through an activation function, which is a non-linear function that bounds the output's range to a known numerical interval. Activation functions are point-wise non-linear functions and typically implemented as a *logistic sigmoid* function $(1/1 + \exp^{-x})$, denoted by σ, and a *hyperbolic tangent* function, denoted by $tanh$. The *hyperbolic tangent* is usually used to implement the block input z_t and the output gate o_t. Computing the output of activation functions can be computationally expensive, and one of the contributions of this work will be to show how it can be accelerated.

Another important feature of the LSTM architecture is the presence of a forward-backward algorithm, known as Connectionist Temporal Classification (CTC) (GRAVES et al., 2006). It is used to process the numeric outputs generated by the network in order to associate them with a discrete label or class. In particular, CTC runs the real-valued outputs of the network through a softmax function, which scales to the range $[0, 1]$ so that they can be interpreted as probabilities; then, CTC uses the resulting information about class probabilities (given the input that was presented) to identify/generate the particular

class that will be used as the predicted output of the network. In the OCR setting, the CTC algorithm is responsible for allowing an LSTM to generate outputs that are directly associated with indices in an alphabet containing all possible characters that may be present in a given input image. In this way, LSTM networks are capable of recognizing entire input sequences without any pre-segmentation or post-processing of their corresponding inputs. They can directly transform the outputs of the network into label sequences because they are trained to predict the conditional probabilities of the possible output labels, given input sequences. The last step is performed by a Translate Back function, which is responsible for identifying, in the alphabet data structure, the target character by an index, and outputs it, one by one, to the output vector of the network creating the predicted string for the given input image.

Even though LSTMs achieve good performance in tasks where a type of memory is necessary, in this work we consider a variant of the LSTM architecture, known as BLSTM. BLSTMs consist of a bi-directional recurrent neural network (as previously defined in Section 2.1.1) composed of many LSTM memory cells. This type of network is composed of two hidden layers: a *Forward Hidden Layer* (FHL) and a *Backward Hidden Layer* (BHL), both of which are connected to a common output layer. The FHL is a layer composed of N LSTM cells that take as input, e.g., an image, and processes its pixel columns from left to right. The BHL, on the other hand, uses a different set of N LSTM cells for processing the same input, e.g., the same image presented to the FHL, and processes its pixels columns from right to left. Each such hidden layer has its own distinct set of weights. The capability of analyzing an image from different perspectives (e.g., by processing its pixels in different orders) results in better accuracy, since it allows the network to take into account information about preceding and following letters when determining which particular character corresponds to the current image location being processed. BLSTMs inherit many of the advantages of LSTMs, such as the capability of preserving the long-term dependencies of its inputs for a longer period of time). In Section 4.1 we will describe characteristics of the particular BLSTM used in this work.

2.2 Optical Character Recognition

As mentioned in Chapter 1, LSTM neural networks often outperform other types of neural networks in terms of accuracy and robustness, when applied to the task of character recognition. For this reason, we choose evaluate the performance of our proposed

optimization techniques (when applied to a baseline BLSTM neural network) in this problem.

The Optical Character Recognition (OCR) problem consists in recognizing printed or handwritten text images; that is, converting printed or handwritten image characters (stored as an image) into machine encoded characters. A text image is usually scanned or taken by a camera from a printed paper data record, such as a passport, receipt, card, book, or other types of documents. The OCR conversion process consists of an analysis of the image, character by character, translating its alphabetic letters, symbols, or numeric digits, into computer character codes that can represent them, such as ASCII codes. This process is an important step of digitizing printed texts in order to make them electronically available, and is widely used in machine processes such as cognitive computing, machine translation, text-to-speech conversion, and text mining.

2.3 String Comparison Algorithms

When evaluating the accuracy of a network tasked with recognizing characters, we need to choose a metric to compare the output produced by the network (a string) and the desired/ground truth output. One possible way of comparing strings is by using string distance algorithms which compute the dissimilarity between two words of arbitrary length.

2.3.1 Levenshtein Distance

The Levenshtein Distance is a metric used to measure the difference between two sequences of characters, or strings. The Levenshtein Distance algorithm measures the difference between two sequences of characters by calculating the minimum number of character additions, deletions, and substitutions required to transform one sequence into the other (Levenshtein, 1966).

The Levenshtein distance between two strings a and b with length $|a|$ and $|b|$, respectively, is denoted by $\text{lev}_{a,b}(|a|, |b|)$ and defined in Equation 2.3. Here, the term $1_{(a_i \neq b_j)}$ is the indicator function, equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $\text{lev}_{a,b}(i, j)$ is the distance between the first i characters of a and the first j characters of b. The first element in the "min" clause of the equation represents the possibility of deletion

(from a to b); the second clause corresponds to the possibility of character insertion, and the third one to the match or mismatch, depending on whether the respective characters are the same.

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1 \\ \text{lev}_{a,b}(i,j-1) + 1 \\ \text{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases} \tag{2.3}$$

2.4 CPU Architectures and Instruction Set Extensions

In our work, we consider using many CPU-specific optimizations, which depend on the particular architecture a neural network is implemented on. In this section we present the CPU architectures used in this work to conduct our experiments and tests. Table 2.1 shows four selected architectures. Xeon E5-2670 is a high-performance CPU targeted at non-consumer workstation/server/embedded systems markets. Core i7-4500U is a power-optimized CPU designed for laptops. The ARM Cortex-A53 and Cortex-A9 architectures are low-power CPUs for embedded devices. They are used by the Raspberry Pi 3 Model B and by Xilinx Zynq-7000 XC7Z045 SoC, respectively. The maximum working frequency of each processor above mentioned is shown in the first column of Table 2.1. The second column of Table 2.1 presents the thermal design power (TDP) in watts of the corresponding processors. TDP is defined as the maximum amount of heat generated by the processor when in typical operation. The number of cores and threads of each processor is presented in the third column of the table; the last column enumerates the extensions to the instruction set of each architecture—which will be exploited in this work when introducing and proposing possible hardware-specific optimizations.

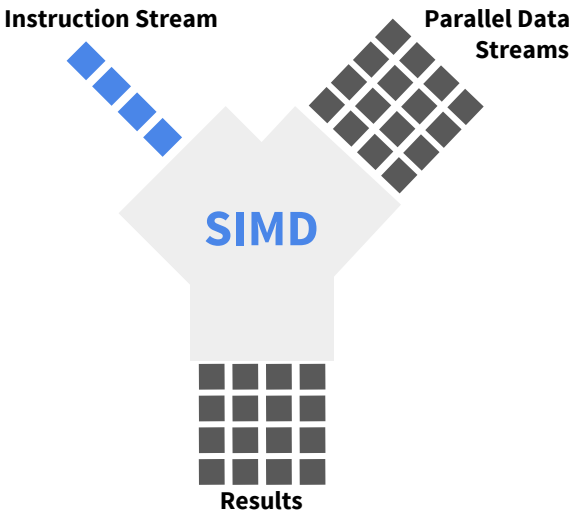
Table 2.1: The reference CPU architectures used in this work

Processor	**TDP [W]**	**Cores (Threads)**	**Instruction Set Extension**
Intel Xeon E5-2670 v3 @ 3.1 GHz (turbo)	90	16(32)	SSE4.2 / AVX2.0
Intel Core i7-4500U @ 3.0 GHz (turbo)	15	2(4)	SSE4.2 / AVX2.0
ARM Cortex-A53 @ 1.2 GHz	<1	4(4)	NEON
ARM Cortex-A9 @ 800MHz	<1	2(2)	NEON

One of the most important optimization techniques applied in this work consists

in exploiting hardware-specific processing units of each architecture. Single Instruction, Multiple Data (SIMD) describes computers with multiple processing units that apply a single instruction to multiple data elements in parallel. Figure 2.4 shows the operating method of these computational units. SIMD processing exploits data-level parallelism. Most modern processors implement SIMD features, but the implementation details vary according to the manufacturer. SIMD operations usually manipulate vectors and are, for this reason, also referred to as *vectorization instructions*.

Figure 2.4: Single instruction, multiple data



Most modern CPU designs also use SIMD instructions to improve the performance of multimedia or intensive calculation portions of the code. Intel architectures provide SIMD through the Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX)—both of which are extensions to the basic x86 instruction set architecture. ARM architectures, on the other hand, provide SIMD through the NEON advanced SIMD architecture extension for the ARM Cortex-A series and Cortex-R52 processors. In the next sections we discuss the specific instruction sets that will be exploited in this work in order to design hardware-specific optimization techniques; in particular, we introduce the SSE, AVX, and NEON instruction set extensions.

2.4.1 Streaming SIMD Extensions (SSE)

Streaming SIMD Extensions (SSE) is a hardware technology that enables a single instruction to process multiple data. It is an instruction set extension over the x86 architecture, designed by Intel. Older processor architectures only processed a single data

element per instruction. SSE enables some special instructions that can handle multiple data elements. This can greatly increase performance of intensive processing applications where a same operation needs to be performed on multiple data objects; e.g. digital signal processing and graphics processing. SSE was designed to replace MMX technology, which is another SIMD Intel technology. It has expanded over the generations of Intel processors to include SSE2, SSE3/SSE3S, and SSE4. Each iteration has brought new instructions and increased performance.

SSE originally added eight new registers of 128-bits, known as XMM0 through XMM7. The 64-bit Intel architectures added a further eight registers, XMM8 through XMM15. Each such register can store four single-precision floating-point numbers of 32-bits or eight short integer numbers of 16-bits as a single data type. Figure 2.5 shows the registers size for the SSE technology.

Figure 2.5: Size of the new registers added by SSE and AVX instruction set extensions

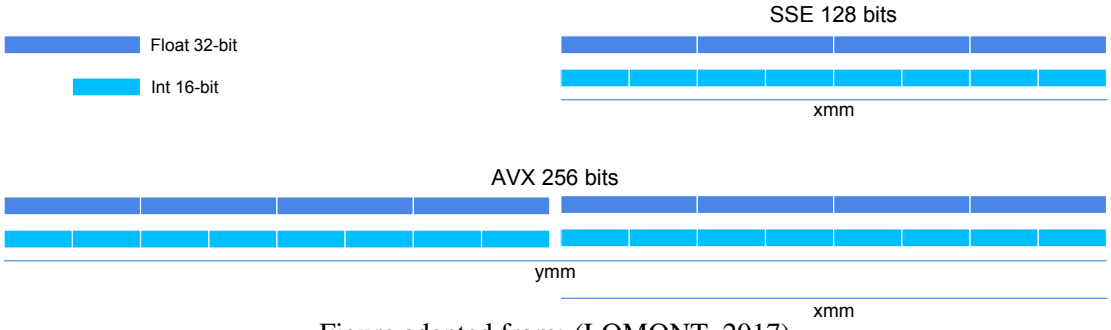


Figure adapted from: (LOMONT, 2017)

2.4.2 Advanced Vector Extensions (AVX)

Advanced Vector Extensions (AVX) are more recent extensions to the x86 instruction set architecture for Intel and AMD microprocessors. AVX provides new features, new instructions, and a new coding scheme. AVX increases the size of SIMD registers from 128 bits to 256 bits and renames them from XMM to YMM. Intel AVX was designed to support 512- or 1024-bits in the future.

Originally, AVX added more sixteen new registers of 256 bits, known as YMM0 through YMM15. Each YMM register can store eight single-precision floating-point numbers of 32-bits or sixteen short integer numbers of 16-bits. In processors with AVX support, the legacy SSE instructions (which previously operated over 128-bit XMM registers) can be extended by using lower 128 bits of the YMM registers, as shown in Figure 2.5.

2.4.3 NEON Advanced SIMD Architecture Extension

ARM architectures also provide SIMD extensions, which are known, in this case, as *Advanced SIMD Extension*, or NEON. The NEON technology is a combined 64- and 128-bit SIMD instruction set that provides standardized acceleration for media- and signal processing applications. The NEON architecture extension was designed for the Cortex-A series and Cortex-R52 processors. It is included in all Cortex-A8 devices, but is optional in Cortex-A9 devices. NEON features a comprehensive instruction set, separated register files, and independent execution hardware.

Similarly to SSE, NEON also supports 8-, 16-, 32- and 64-bit integer and single-precision (32-bit) floating-point data and SIMD operations for handling intensive processing applications. NEON can also process four single-precision floating-point numbers of 32-bits or eight short integer numbers of 16-bits as a single data type. Devices such as the ARM Cortex-A8 and Cortex-A9 support 128-bit vectors but process 64 bits at a time, whereas newer Cortex-A15 devices can process 128 bits at a time.

3 RELATED WORK

In this chapter we review related work that also aims at using different types of optimization techniques to accelerate neural networks. As previously mentioned, some of the optimization techniques presented in this document, such as better memory allocation, different parallelization schemes, loop unrolling, and intrinsic functions have been previously used to optimize different applications; in this work, we focus on how they can be used in isolation or combined in order to improve one particular complex type of recurrent neural network. In this chapter we introduce and discuss existing hardware implementations of neural networks which are good representatives of how developers typically convert existing software implementations to dedicated hardware, in order to exploit low-level features and improve performance. We also discuss previous works that speed up neural networks by using software-only optimizations, and works that combine software optimizations and hardware capabilities of a given target architectures where the NN will be deployed. As will be discussed in what follows, many works aim at optimizing simpler neural network architectures, but few focus on optimizing recurrent neural networks such as BLSTMs.

3.1 Accelerating Neural Networks via Dedicated Hardware

Many researchers have proposed to use dedicated hardware (e.g. FPGAs) to speed up neural networks. However, this approach is expensive and time-consuming, since it is necessary for an expert to migrate a software implementation to a hardware implementation; the approach is also not flexible in that future modifications to the network are harder to be implemented, since a modification to the hardware is needed. When this approach is possible, however, the resulting implementation offers considerable runtime and power consumption reduction. In this work we focus on the particular case where these assumptions cannot be met and the developer may choose, by contrast, to deploy software optimizations and to exploit hardware capabilities in order to speed up an NN. In this section, we review a few existing works that aim at accelerating neural networks via dedicated hardware implementations.

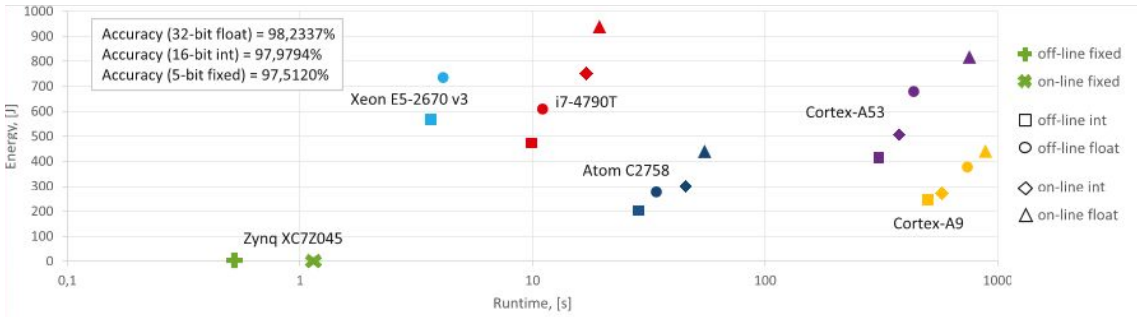
The use of dedicated hardware (e.g. FPGA) to speed up Convolutional Neural Networks (CNNs) is proposed in works, such as (ZHANG et al., 2015), (QIU et al., 2016) and (MOTAMEDI et al., 2016). (ZHANG et al., 2015) implements a CNN ac-

celerator on a VC707 FPGA board, and analyzes its computing throughput and required memory bandwidth after using various optimization techniques, such as loop tiling and loop transformation. (QIU et al., 2016) proposes a CNN accelerator to be deployed on an Xilinx Zynq ZC706 board for use in the Image-Net large-scale image classification task. It uses a dynamic-precision data quantization method, a convolution design, and a better data arrangement method to achieve state-of-the-art performance. (MOTAMEDI et al., 2016) proposes an accelerator that can effectively leverage all of the available parallelism sources to minimize the execution time of a neural network. All of these works have in common the use of three main optimization techniques that we also consider: the use of all parallelization methods available, better memory allocation and access, and the reduction of the amount of memory required by the NN. These approaches also use specific low-level optimizations methods available in FPGAs.

The work of (RYBALKIN et al., 2017) proposes a hardware implementation of a BLSTM neural network for optical character recognition. They show that this computationally intensive visual recognition task benefits from being migrated to a hardware accelerator, outperforming high-performance CPUs in terms of runtime, while consuming less energy than low-power systems with negligible loss of recognition accuracy. The baseline software implementations used in our work, which we will optimize via a set of proposed techniques, is the same one used by (RYBALKIN et al., 2017) to construct their dedicated hardware. All of the optimizations used to improve the runtime in their hardware accelerator were also considered when designing our optimization techniques; ours, by contrast, *do not* require the neural network to run in a dedicated hardware. In (RYBALKIN et al., 2017), the authors designed a hardware architecture capable of implementing a BLSTM in FPGA; in our work, we will optimize (via many specific software *and* hardware optimization techniques) that same baseline software. A comparison between their dedicated hardware and the same software baseline that we use in this work is shown in Figure 3.1. Note that the analyses presented in (RYBALKIN et al., 2017) compare only with a partially optimized version of the BLSTM. In this work, we propose and evaluate other combinations of optimization techniques that can also be used to further improve this neural network implementation. See Chapter 4 for more details.

As can be seen in Figure 3.1, depending on the target architecture where the neural network is deployed, a dedicated hardware architecture can provide runtime acceleration in the range of 5 to more than 600 times, when compared to a baseline software implementation with simple optimizations. Although these results are impressive, this kind of

Figure 3.1: Runtime and energy consumption comparison between a dedicated-hardware and optimized software implementations of a BLSTM



hardware-only approach has disadvantages with regards to the expertise required to implement an NN in hardware; furthermore, hardware-implemented NNs are not flexible in terms of how easy it is to perform updates to them or to the algorithm that trains them. The experiments results we present in Section 6.3 show that by deploying novel optimization techniques, it is possible to decrease the gap between the performance achievable via a hardware-dedicated architecture (such as in (RYBALKIN et al., 2017)) and what can be achieved by combining software optimizations and by exploiting hardware capabilities.

3.2 Accelerating Neural Networks via Software-Only Optimizations

Software-only optimizations are attractive because they are simpler to deploy and more people are capable of performing them, but they do not exploit all resources available that could lead to improvements to a neural network, such as specific hardware instructions that implement linear algebra operations (SSE, AVX and NEON).

It is not common people to focus purely on software optimizations, while leaving aside the architecture where the neural network will be executed. When a software-only approach is used, the optimizations can be of two types: *1)* general-type optimizations, which could be applied to any software (e.g. loop unrolling, parallelization via threads); or *2)* optimizations that take into account the sequence of computations required to produce the output of a particular type of neural network, given an input. Optimization techniques of the second type exploit the fact that it is possible to implement the functionality of a neuron through the dot product of inputs and weights, or, equivalently, through linear algebra functions where the inputs and the weights are no longer interpreted as sets of scalars, but as vectors. The advantage of this second interpretation is that developers often have access to scientific computing libraries that implement extremely efficient ver-

sions of linear algebra operations in software. Most available implementations of neural networks make use of such software-based optimizations.

In this work, we also explore these commonly-used software optimizations, but we extend the set of optimizations to be deployed with other software-based acceleration techniques. We observed in our literature review, for example, that hardware-based optimization methods often speed up processing *i)* via the reduction of numerical precision (e.g. floating-point parameters are transformed and processed as short integers or via a fixed-point representation); and *ii)* by using lookup tables designed specifically to deal with the problem of computing activation functions of neurons. In this work, we discuss (in Sections 4.3.6 and 4.3.7) ways to implement such optimizations at the software level as well. We also propose software optimizations that are designed specifically for the particular baseline neural network used in this work; we do so, for instance, by allowing two types of parallelization (at the input level and at the level of neurons), thereby effectively calculating the various intermediate outputs of a network simultaneously.

3.3 Accelerating Neural Networks via Software Optimizations and Hardware Capabilities

In cases where implementing a dedicated hardware is not possible, developers may wish to combine software-based optimization techniques while also exploiting hardware capabilities that are architecture-dependent. This is the type of approach that we adopt in this work, in great part because it allows, in our opinion, for a better trade-off between the performance levels that can be achieved (via software and hardware optimization techniques) while still not requiring one to fully re-implement a baseline software in dedicated hardware.

Some techniques to reduce the computational costs of neural networks on x86 CPUs are discussed in (VANHOUCKE; SENIOR; MAO, 2011). Loop unrolling, data layout, memory allocation, use of SSE instructions, among other improvements, are discussed in that article. Many of the improvements proposed there are used in this work, but via approaches that do not require a dedicated hardware. (VANHOUCKE; SENIOR; MAO, 2011) evaluates their hardware on a speech recognition task, showing that a real-time hybrid Hidden Markov model/neural network (HMM/NN) can be built and produce a 10x speedup over an unoptimized baseline software, and a 4x speedup over an aggressively optimized floating-point baseline software while undergoing no costs in terms of

accuracy. We extend that work in many ways, by proposing *i)* novel optimizations for software implementations, such as lookup tables, AVX and NEON intrinsics; and, *2)* by showing how to deploy some of their hardware-dependent optimizations in different architectures, and by adjusting them to a different and more complex neural network architecture.

(COLLOBERT; KAVUKCUOGLU; FARABET, 2012) introduces a new framework, called Torch7, which is especially suited to achieve a high computational performance. To do so they provide float or double representation for the neural networks parameters, memory allocation control, ordered accesses to memory, use SSE or NEON instructions when possible, and support two ways of parallelization: OpenMP and CUDA. These authors mention that GPUs (e.g. running with CUDA) are often not as attractive in practice as most could have expected: GPU-specific implementations of a neural network may require heavy extra work for a speedup to be achieved, which can be significant when compared to the performance that can be achieved in simpler ways—e.g. with just a few extra lines of code with OpenMP for developing parallel applications on CPUs.

(APPLEYARD; KOCISKÝ; BLUNSOM, 2016) describes three stages of optimizations that be incorporated into a BLSTM neural network running on GPUs. Firstly, the optimization of a single cell, secondly the optimization of a single layer, and thirdly the optimization of entire network as a whole. They combine matrix operations that share a same input into a single larger matrix operation—one of our first optimization techniques that we also evaluate. The first type of optimization they propose to apply to a single cell corresponds to increasing the parallelism of a single RNN cell. The second improvement proposed to a single cell corresponds to the fusion of point-wise operations—because of their independent nature, it is possible to fuse all the point-wise kernels into one larger kernel. The primary strategy used in this latter case was to expose as much parallelism to the GPU as possible in order to maximize the use of hardware resources. As previously mentioned, the use of GPUs has a few advantages (a larger number of processing units), but it is also more complex to deploy, and the architecture is not always available in simpler devices. Furthermore, the overhead of launching GPU kernels, and of transferring data between CPU and GPUs, can sometimes be hard to amortize when using small neural networks.

4 OPTIMIZATION TECHNIQUES FOR NEURAL NETWORKS

In this chapter we propose and discuss neural network optimization methods that are based both on software optimization techniques and/or that exploit hardware capabilities of the particular architecture where the network will be deployed. First, we discuss the baseline BLSTM implementation that will be improved upon in our work. We also discuss how we identified the main runtime bottlenecks of that baseline, before applying the optimizations that are proposed here. Finally, we introduce all proposed techniques explored in this work, explaining each one in details, showing their benefits, trade-offs, and discussing how to apply them.

4.1 Baseline BLSTM Implementation

The baseline BLSTM software implementation that we will optimize in this work is presented in this section; many of our optimization design decisions were made in order to exploit particular characteristics of this software. Every optimization that we propose in this section will be compared (in Chapter 6) to this baseline version of the neural network. The baseline software was originally developed in C++ and it is a straightforward implementation of a BLSTM NN. In (RYBALKIN et al., 2017), this baseline implementation (un-optimized) was compared to an FPGA hardware implementation of the same network, as previously discussed in Chapter 3.

A set of pre-trained network parameters (weights) and an image testing set composed of text images were computed and made available for use by (RYBALKIN et al., 2017), and will be used later on when evaluating our optimized BLSTM NN in the task of Optical Character Recognition. The baseline implementation of the BLSTM makes use of single-precision floating-point format to store weights and input images that must be processed. It is composed of three main parts, or steps, which are executed sequentially: *1)* allocation and loading of data structures; *2)* BLSTM NN processing; and *3)* metrics gathering. We explain each one of these steps in what follows.

4.1.1 Step 1: Data Structure Allocation and Loading

The first step is responsible for allocating and loading all the testing set input images and ground-truth strings from the hard drive (HD) to memory. All the other data structures used during the NN processing are also allocated and loaded to memory at this part of the code. These other data structures are the weights used by the NN to appropriately represent a mapping function from inputs to outputs, passing through the hidden layer (see Section 2.1 for more details), as well as the alphabet needed to transform the output probabilities of the NN into characters.

4.1.2 Step 2: Computing Neural Network Outputs

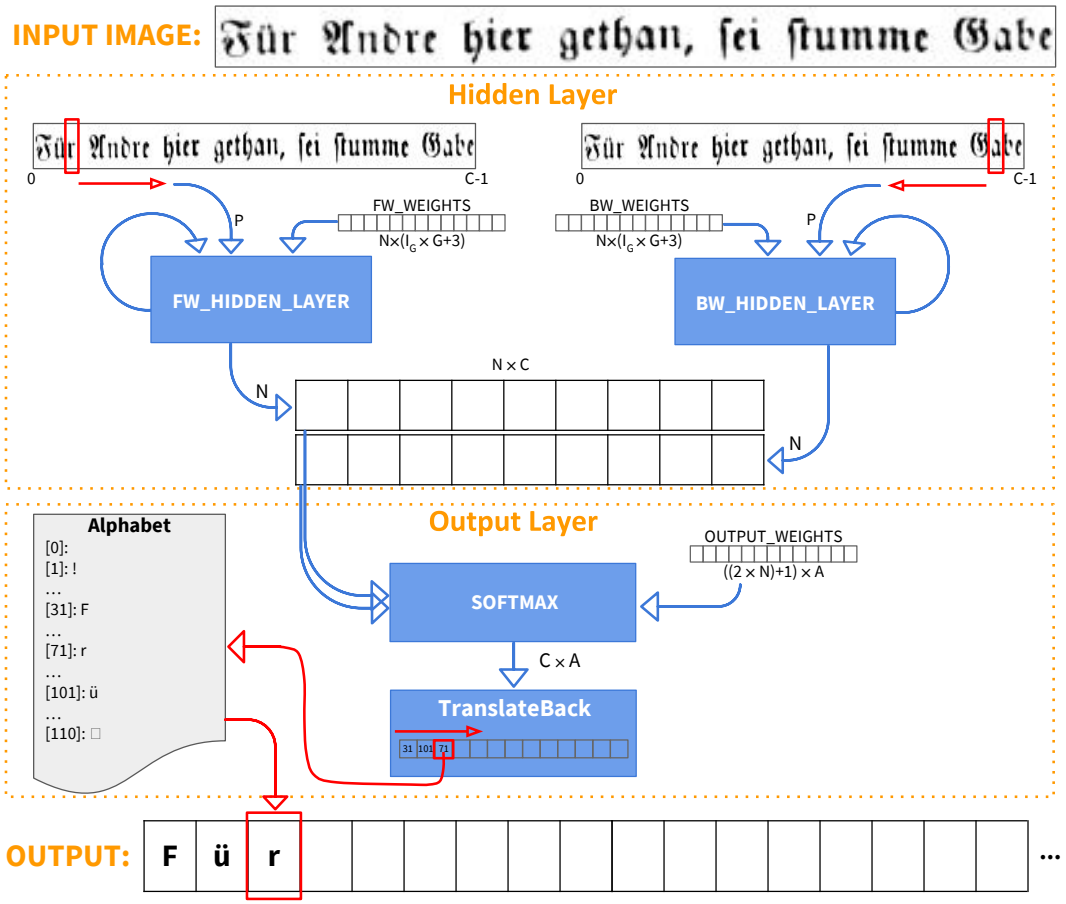
The second step is the NN processing itself, i.e., the computation of its outputs. Figure 4.1 presents the implementation details of the BLSTM neural network with all the main functions and NN processing steps. This is the most important part of the code, since this is the portion of the software that we aim at speeding up in this work. Table 4.1 introduces important information and parameters used by the baseline implementation.

Table 4.1: General information about the baseline BLSTM implementation

Number of images in the testing set, T	3401
Max number of columns per image, C	732
Number of pixels per column, P	25
Number of LSTM memory cells in a hidden layer (neurons), N	100
Number of gates including block inputs to a memory cell, G	4
Number of inputs per gate, I_G	126
Number of alphabet symbols, A	110
Number of inputs per output unit, S_O	201

A testing set of T images, corresponding to text lines, is used as input to the NN. The baseline network implementation includes an input layer that receives these images, column by column. Each input is represented as a grayscale image that is P pixels height and that has a variable length of up to C columns. The BLSTM network is composed of a Forward Hidden Layer (FHL) and a Backward Hidden Layer (BHL). As discussed in Section 2.1.2, an FHL is a layer composed of N LSTM cells that takes as input, e.g., an image, and processes its pixel columns from left to right. A Backward Hidden Layer (BHL), on the other hand, includes a different set of N LSTM cells for processing images from right to left. Each such hidden layer has its own distinct set of weights. These

Figure 4.1: BLSTM implementation.



weights were previously loaded to memory by the first step of the code, and the loaded weights are passed as parameters to the corresponding hidden layer. Within the FHL and BHL, we have the LSTM cells, or neurons, of the NN. Each LSTM gate receives I_G as inputs a set of *source values*: a single bias value, a set of P pixels of an image column, and N output values received via the recursive connections of the network (originating from the neuron's activation at the previous time step). The LSTM cells process their source values and weights by using dot products, vector multiplications, vector concatenations, and activation functions; see Section 2.1.2 for more details about the LSTM cell processing.

As a result of the hidden layers processing, two output vectors of probabilities are generated, one with the output of the forward processing, and other one with the output of the backward processing. In order to use this network as a classifier, we make use of a CTC layer. The outputs of the hidden layers are normalized using a softmax function and set to the range (0, 1). The normalized outputs from the softmax layer are then used by the Translate Back function to estimate the conditional probabilities of a given label (character) in the Alphabet (or a blank) at time t during the processing of a given input

image. The Translate Back function identifies, in the Alphabet data structure, the target symbol encoded by the probabilities vector, and outputs the corresponding characters, one by one, thereby creating the string output for the given input image.

4.1.3 Step 3: Metrics Gathering/Performance Evaluation

The last step is responsible for gathering important metrics to be used as a way of comparing the different configurations of the software, in terms of which subset of optimization techniques are deployed. Accuracy, runtime and energy consumption are the main metrics used in this work to evaluate our optimizations—see Section 5.3 for more details. Later, in chapter 6, we will present the results obtained after the application of the software optimization techniques and hardware capabilities explored in this work.

As soon as the last testing set image is processed by the NN, a function computes the recognition accuracy of the results based on the Levenshtein distance with respect to the testing set ground truth; see Section 5.3. The runtime of *step 2)* alone is then computed and printed to the user. Therefore, the default information shown to the user of the software is the accuracy and the runtime of the network. A debug flag allows it to output the text lines, as produced by the network, the corresponding testing set ground truth strings, and the number of necessary computational operations to transform the output of the NN into the ground truth string. The software output when running the BLSTM neural network with this debug flag enabled is exemplified in Figure 4.2.

Figure 4.2: Example of the BLSTM execution and output

```
0
und Gegensätze ihres Charakters nebeneinander in Gleichkraft
und Gegensätze ihres Charakters nebeneinander in Gleichkraft

0
erhielten, während beispielsweise bei Schinkel und Winkelmann
erhielten, während beispielsweise bei Schinkel und Winkelmann

0
das Griechische über das Märkische beinah vollständig siegte. Bei
das Griechische über das Märkische beinah vollständig siegte. Bei

1
drlückte oder beherrschte das andre und die Neu-Uniformirung eines
1drückte oder beherrschte das andre und die Neu-Uniformirung eines

Accuracy: 98.2337%
Measured time ... 248887 milliseconds
```

Since our goal is to speed up a software implementation of the NN, one of the first things to do is to identify runtime bottlenecks by making use of profiling tools; that is, by collecting runtime measurements of the different functions and instructions to find the

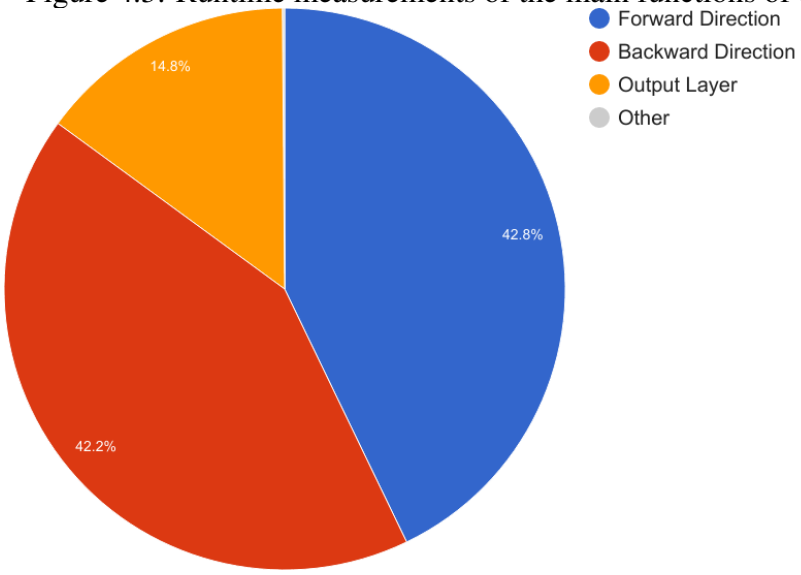
most intensive/expensive computational parts of the code. In the next section, we show the first runtime measurements of the main functions of the code, identified via the C++ <*chrono*> library.

4.2 Identifying Bottlenecks in the Baseline BLSTM Implementation

Imagine a hypothetical application which spends 80% of its runtime in a function that represents just 5% of the code. An improvement of 50% in runtime on this small part of the code would generate a general improvement of 40% of the application runtime. Identifying bottlenecks of a software is very important and useful since we desire to maximize the impact of any optimization steps by applying them to the most computationally expensive parts of the code.

The aim of this section is to identify the bottlenecks of the BLSTM processing procedure described in step 2. The main parts of the NN code, with large runtime, will be measured and evaluated in order to decide if they deserve special attention when implementing improvements. We start the bottleneck identification procedure by applying runtime measurements to the main functions of the code. The results of these measurements can be seen in Figure 4.3. Such measurements appear to suggest that the main bottlenecks are inside the Forward and Backward hidden layer functions, since together they represent 85% of the NN runtime.

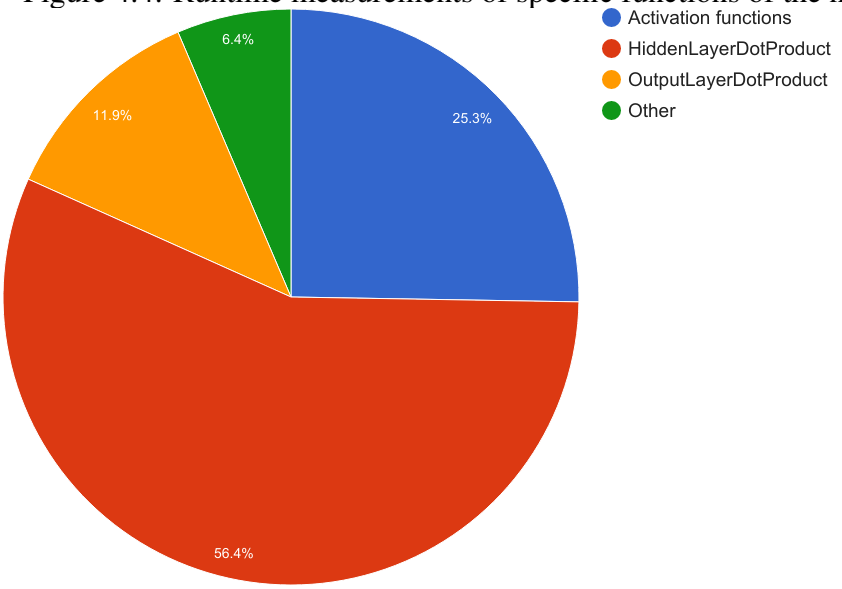
Figure 4.3: Runtime measurements of the main functions of the code



After this first look at the data, we applied more specific runtime measurements, in specific parts of the code, such as the neuron function, which consists of the implemen-

tation of complex LSTM cells and their processing. Figure 4.4 shows these additional measurements. A closer look at the collected data indicates that we have as main bottlenecks the dot products between source inputs and weights of the LSTM, and that these have a direct impact on the total BLSTM runtime.

Figure 4.4: Runtime measurements of specific functions of the neurons



The most expensive computations of the NN are the dot products, responsible together for more than two-thirds of total runtime. The data yielded by the chart of Figure 4.4 provides strong evidence that the optimization techniques must be applied at least at a first moment in the crucial dot products between the inputs and weights of the neural network FHL and BHL. Other important part of the code that must have our focus for improvements is the output layer dot product inside the softmax function. However, it is still interesting to notice and keep in mind that one quarter of runtime is being used by the neuron activation functions and other neuron operations. This part of the code involves much more operations and complexity, hence it will not be the focus for our first optimization techniques. Now we have analyzed the bottlenecks lets move on to the optimization techniques we have adopted to speed up the runtime of the NN.

4.3 Software Optimization Techniques

Having identified the main bottlenecks of the baseline neuron network, we now introduce the main optimization techniques that can be applied to the software, and that are architecture-independent. We discuss, in what follows, the two parallelization scenarios that can be used to split the work of the network among the available computational

units. We start by introducing some general and important software-based optimization techniques that can be used, such as hidden layers aggregation, better memory allocation/access, and dot product loop unrolling. After that, we evaluate the impact of replacing simple loop unrolling by the use of an optimized linear algebra library for the dot products; however, since this is one of the bottlenecks of our application, as shown in the previous section, we also propose other, more aggressive optimization techniques for this problem (see Section 4.4). At the end of this section, we introduce two optimization techniques which allow a trade-off between NN accuracy and NN performance: one that reduces the amount of data that needs to be manipulated during the most intensive parts of the code, and another that changes the way expensive neuron operations are implemented.

4.3.1 Parallelization Scenarios

One of the simplest techniques to exploit CPU hardware capabilities is the use of parallelization or threads. To use these, we first need to make a non-trivial decision: to define which parts of the code must be parallelized. One possibility, in case of a testing set of T images, is to process each image in parallel using T threads. Therefore, each thread corresponds to the execution of a whole BLSTM NN to process a specific input image. We call this *input-level parallelization*. Another possibility is to parallelize the set of computations needed to determine the output of the NN when applied to a given input. In particular, it is possible to parallelize the computation of the individual neurons (LSTM cells) of the hidden layer. In this case, we have one thread per neuron of the NN. We call this *neuron-level parallelization*. The creation of the threads, in this work, was made via the Open Multi-Processing (OpenMP) API. We chose this API since it provides a portable, scalable, simple, and flexible interface for parallel development for many different platforms, ranging from the standard computers to supercomputers. The following sections discuss these two possible parallelization scenarios in more details.

4.3.1.1 Input-level Parallelization (Offline)

The first parallelization scenario described above is better suitable for batch processing, since the parallelization is applied in order to process separate images at different points in time—see Figure 4.5. We call this scenario *input-level parallelization*, or just "Offline". In this approach, there is one thread implementing the entire neural network,

and it used to process one given input image. Many threads/NNs are run in parallel, for different images, and the output of each thread is stored in a data structure collecting all the output strings. The code presented in Figure 4.6 shows how we can create such threads in order to process different input images in the testing set.

Figure 4.5: Input-level parallelization

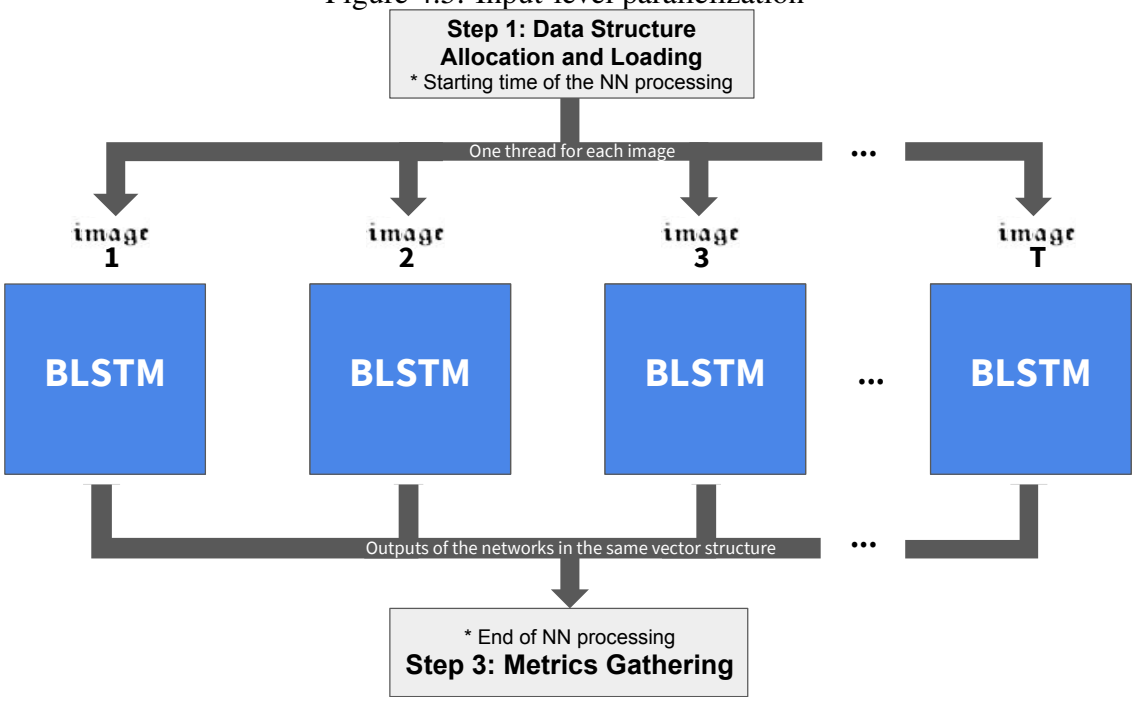


Figure 4.6: Threads creation example with OpenMP

```
1    ...
2    #pragma omp parallel
3    #pragma omp for schedule(dynamic)
4    for(unsigned int i = 0; i < vecInputImage.size(); i++){
5    ...
```

This is an interesting approach for use when a batch of images are available in memory for processing, which allows the NN to perform the dot products of all the chain of LSTM cells before to use them inside the LSTM computations. Since we start presenting an image to the network, we can compute the dot products between the corresponding inputs and the weights of all neurons in timestep t, storing the resulting intermediate outputs for posterior processing. Other important feature of this parallelization scenario is the possibility to use a contiguous and big portion of memory to the weights (network parameters), benefiting from memory locality advantages that will be discussed in Section 4.3.3.

There are few possibilities to parallelize BLSTM NNs internal computations, since the architecture of this network requires the computation of all outputs at a given t, and only then, in a synchronous way, use them as inputs to the next time step. This is generally true except for some specific computations, such as the LSTM dot products mentioned above.

4.3.1.2 Neuron-level Parallelization (Online)

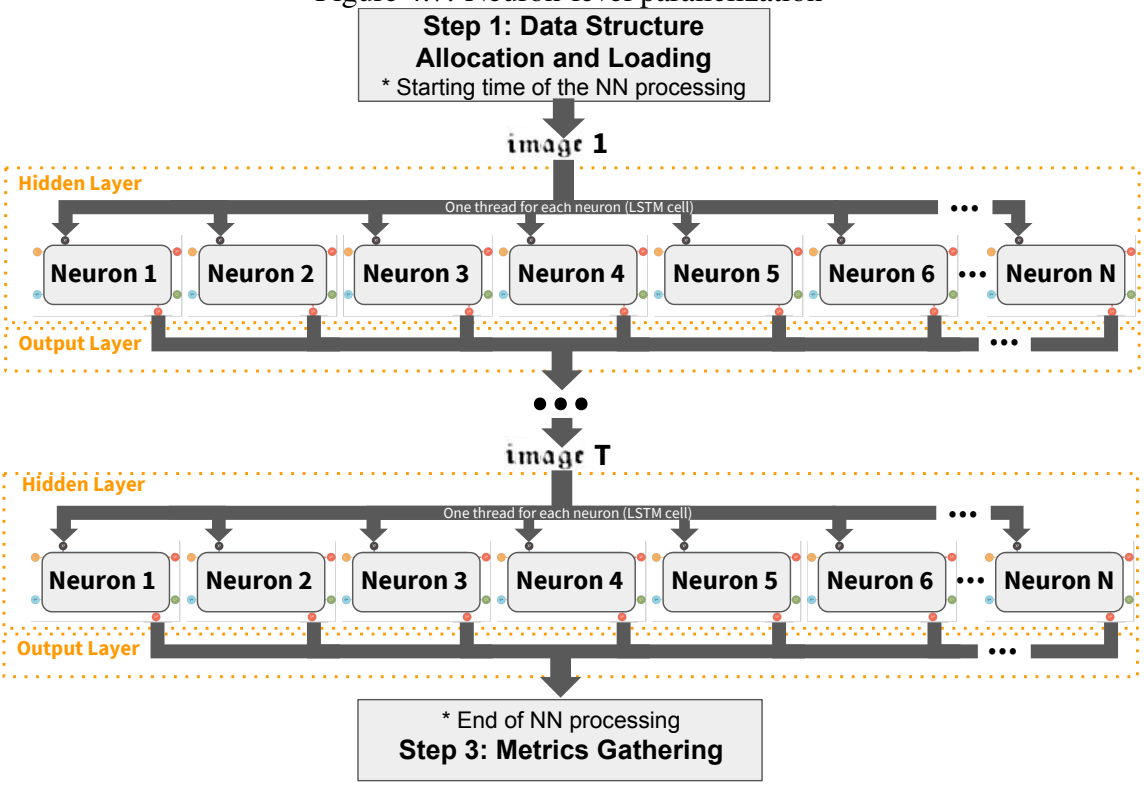
Another parallelization scenario can be advantageous when input images appear one at a time and can only be read incrementally, instead of being completely pre-loaded in memory. In this case, the allocation of a whole NN resource at the beginning of the processing step of a given input image is not the best alternative, since the NN does not have the entire image available for use. Therefore, the parallelization must be applied for each neuron inside a LSTM cell and no dot product pre-processing can be performed as previously suggested for the input-level parallelization. When implementing neuron-level parallelization, the software only allocates resources which will be used, which in some cases is the best option given an energy budget.

In this second parallelization scenario, we parallelize the network at the level of neurons, meaning that each thread processes separate parts of the internal computation of a neural network at a given point in time. All threads' outputs are then composed to form the output of the neural network for one given input image. This output is stored in an intermediate data structure to be compared to the ground truth, providing the accuracy metric. We call this scenario neuron-level parallelization, or just "Online" because of its application (i.e. real-time OCR). In short, we have in this scenario one image being processed at a time and the neural network's output for a given image is obtained by composing N threads, one per neuron; see Figure 4.7 for an example of this type of parallelization.

4.3.2 Hidden Layers Aggregation

Since the most expensive functions (dot products and activation functions) are split in two different hidden layers, one possibility would be to join them and take advantage from operations that could be reused, or from data aggregation. In particular, instead of processing two hidden layers separately, we aggregate the data used by them in order to

Figure 4.7: Neuron-level parallelization



reduce the number of allocated resources, at a cost of increased data structures, i.e. instead of using 8 vectors to store the network weights, we start using just one big vector.

The use of hidden layer aggregation implies that it is possible to process input images and weights with forward and backward data interleaved. This step is closely related to the next section of this work, in which we show how to rearrange the memory allocations that are passed as arguments to these two separate hidden layers putting them together. As result of this change, it is possible to compute the NN output as if it had only one hidden layer, containing all the data but with doubled number of operations.

4.3.3 Memory Allocation and Access

One of the most basic principles of high-performance computing is that once a given memory address is read, nearby memory addresses get loaded into the various caches on the processor. This makes nearby data available to the CPU much faster than it would be if it had to fetch it from memory. The most immediate consequence of this fact is that one should strive to have most data necessary for any given computation walk stored in contiguous memory.

With this in mind, and knowing that the dot product computations between weights

and inputs are the bottleneck of our application, we start by aggregating all the dot product functions interleaving their data structures, in order to have better cache locality performance. Initially, the dot product functions performed by the neurons were processed separately by four different functions, one for each gate. Each function was executed twice since there were two hidden layers, totaling eight dot product function calls per neuron. We change this part of the code to execute a single function that could compute the eight necessary results per neuron. The hidden layer of the network is changed to receive just one large vector of weights containing the data of all the separated weights used before this modification. Instead of four weights being accessed twice, we create this vector of weights with the forward and backward layers weights interleaved, in order of faster access by the new single dot product function. Other data structures used by the NN are also rearranged to occupy contiguous memory addresses, so that they can be accessed in sequence, thereby avoiding jumps between the data addresses needed to perform some computation.

The four different weights mentioned above are the WGI, WGF, WGO and WCI (responsible to control the input, forget, output and block input gates, respectively). They are aggregated into a single vector WS; see Figures 4.8 and 4.9 to compare the different ways one can allocate the weights.

Figure 4.8: Allocation of the weights used by the LSTM cells dot products before weight aggregation

```
1    WGI = new float [NUMBER_OF_NEURONS * NUMBER_OF_INPUTS];
2    WGF = new float [NUMBER_OF_NEURONS * NUMBER_OF_INPUTS];
3    WGO = new float [NUMBER_OF_NEURONS * NUMBER_OF_INPUTS];
4    WCI = new float [NUMBER_OF_NEURONS * NUMBER_OF_INPUTS];
```

Figure 4.9: Weights aggregation in a single vector, instead of using four different weights for each hidden layer

```
1    WS = new float [NUMBER_OF_HIDDEN_LAYERS * NUMBER_OF_GATES *
2    NUMBER_OF_NEURONS * NUMBER_OF_INPUTS]
```

Another important aspect related to memory allocation for higher performance results is memory alignment. Memory alignment means putting data at a memory address that is some multiple of the word size of the CPU, which can increase performance of the system due to the way CPU handles memory. To align the most important data vectors we reallocate every vector whose size is not a multiple of 16 bytes to the minimum size

necessary to make it a multiple of 16. This implies in using zero padding: inserting zeros between the last value stored in the vector and the lowest value for the valid word size. In the OCR application we consider in this work, the number of inputs given to the network (NUMBER_OF_INPUTS in code) is 126, represented by I_G in Table 4.1. The lowest value multiple of 16 that is larger than 126 is 128, so a new definition, called NUMBER_OF_INPUTS_SSE, is created and used instead of NUMBER_OF_INPUTS, and every data structure using this definition had two additional 0 values added at their ends. For linear operations, such as dot product and matrix addition and multiplication, the embedding of these zeros do not affect results. More reasons for these modifications will be explained in details in Section 4.4.

4.3.4 Loop Unrolling

Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize the program execution speed at the expense of its binary size, which corresponds to a space-time trade-off. This transformation can be performed manually or via an optimizing compiler.

The goal of loop unrolling is to decrease the runtime needed for a loop iteration by reducing or eliminating instructions that control the loop, such as pointer arithmetic and "end of loop" tests at each iteration, thereby reducing branch penalties and hidden latencies, such as delays to read data from memory. To eliminate such computational overhead, we re-write all dot product loops in our application so that they become repeated sequences of similar but independent statements, as shown in Figure 4.10. With the multiple accumulators being updated in parallel, the compiler has more freedom to pipeline operations and distribute them across different floating-point units.

We introduce in Figure 4.10 an example of the use of loop unrolling in a dot product function. This technique can be easily deployed and is used by compilers to optimize loops. However, as discussed in Section 4.3.3 we changed the code to perform dot products in just one function, and in this case, the implementation of loop unrolling for the dot product function is much more extensive and complex than the example provided. The gcc compiler do not identify that it could apply loop unrolling to the dot product function of the baseline implementation, even if provided special compilation flags such as -O3, which forced us to implement this technique manually. Using -Ofast, gcc compiles using some technique to better run the loops of the code, nevertheless, the results are not

Figure 4.10: Loop unrolling example

```
// dot product function between inputs and weights without any optimizations
inline float DotProduct(float *source, float *weight){
    float sum = 0.0;
    for(i = 0; i < NUMBER_OF_INPUTS; i++){
        sum += source[i] * weight[i];
    }
    return sum;
}

// dot product function between inputs and weights with loop unrolling
inline float DotProduct(float *source, float *weight){
    float sum = 0.0;
    for(i = 0; i < NUMBER_OF_INPUTS; i+=4){
        sum0 += source[i+0] * weight[i+0];
        sum1 += source[i+1] * weight[i+1];
        sum2 += source[i+2] * weight[i+2];
        sum3 += source[i+3] * weight[i+3];
    }
    return sum0 + sum1 + sum2 + sum3;
}
```

comparable with the technique applied directly in code.

4.3.5 Use of High-Performance Libraries

One of the first things to consider when optimizing software performance is to use optimized code designed by expert researchers/professionals who spend considerable effort creating solutions that achieve best performance. To compare with our optimization techniques to solve the dot product function of the BLSTM, we create a configuration of dot product function making use of Eigen BLAS package— see (EIGEN..., 2017) for more detailed information about this library.

Eigen is a C++ template library for linear algebra, and includes functions to process matrices and vectors. It is versatile, fast and reliable. Optimized dot product functions are provided via the functions adjoint() or dot(), whose use when implementing the dot products required by a neural network is depicted in Figure 4.11. Internally, SIMD instruction set operations and cache optimizations are used to provide the best possible performance for the target architecture.

Even though this library is easy to use, efficient and portable, higher performance can be achieved if custom optimization techniques can be applied to the code, even if at

Figure 4.11: Eigen dot product example

```
1
2   #include <Eigen/Eigen>
3   using namespace Eigen;
4
5   ...
6
7   void DotProduct(float *source,
8                   float *WS,
9                   float *outputs){
10
11      Map<VectorXf> pSource(source,NUMBER_OF_INPUTS_SSE);
12      Map<VectorXf> wgi(WS,NUMBER_OF_INPUTS_SSE);
13      Map<VectorXf> wgf((WS+NUMBER_OF_NEURONS*NUMBER_OF_INPUTS_SSE)↩
            ,NUMBER_OF_INPUTS_SSE);
14      Map<VectorXf> wgo((WS+2*NUMBER_OF_NEURONS*↩
            NUMBER_OF_INPUTS_SSE),NUMBER_OF_INPUTS_SSE);
15      Map<VectorXf> wci((WS+3*NUMBER_OF_NEURONS*↩
            NUMBER_OF_INPUTS_SSE),NUMBER_OF_INPUTS_SSE);
16
17      outputs[0] = pSource.adjoint()*wgi; //pSource.dot(wgi);
18      outputs[1] = pSource.adjoint()*wgf; //pSource.dot(wgf);
19      outputs[2] = pSource.adjoint()*wgo; //pSource.dot(wgo);
20      outputs[3] = pSource.adjoint()*wci; //pSource.dot(wci);
21
22  }
```

To make use of Eigen library and enable vectorization—see Section 4.4 for more details,—it is necessary to call the functions whenever you want to have optimized performance and compile the software with the correct flags to enable the SIMD instruction set operations intended. We make use of (-Ofast) and (-msse4.2 or -mavx2 or -march=native) flags for Intel architectures and (-Ofast) and (-mfpu=neon) flags for ARM architectures.

the cost of developing time, complexity and precision reduction. In the next section, we discuss one of these possible custom optimization techniques to further improve runtime and improve performance.

4.3.6 Precision Reduction

Dot products performed in the hidden and output layers of a neural network operate over network weights and are responsible for the largest part of required memory bandwidth. Furthermore, the vector data structures are responsible for the highest memory resource consumption. Thus, these vectors are one of the top priorities for optimizations. The baseline implementation of the software uses single-precision floating-point representation to store and compute weights, which means that each number occupies 4 bytes (32 bits) in memory. We reduce the precision of the network parameters by using

short integers of 16 bits both for its weights and inputs. This new configuration of the optimized neural network is referred to in what follows as *INTEGER IMPLEMENTATION*, while the previous configuration of the software, with no precision reduction, is referred as *FLOAT IMPLEMENTATION*. In Chapter 6, we show that a negligible accuracy reduction is incurred if using the integer implementation, in comparison to the performance of the original float implementation.

To make use of this technique, both weights and image inputs are first represented as float variables and scaled by taking the maximum value of each data vector and normalizing them to the [-128,127] range. The code that performs this scaling procedure is shown in Figure 4.12. In this work, we empirically selected the coefficients to be used in the scaling process according to the maximum and minimum values found in the weights and images dataset used to our OCR problem.

The vectors used in the NN processing are declared as short integers and use 16 bits to each position. Whenever the previous floating point data would be assigned to a vector position, it is first multiplied by the coefficient pre-defined at the begin of the code. After this process, vectors are used normally by neural network except by the fact that they must be treated as integers. The conversion from integers back to floating point is made at the end of the processing of dot product functions, since the multiplication between two 16-bit integers results in an integer value of 32 bits, thus making possible a direct conversion to the previous floating-point representation.

We observed, during our literature review, that this precision reduction technique is commonly used when designing dedicated hardware implementations, but not so much when seeking to construct optimized networks via pure software-based optimizations or methods that exploit particular hardware characteristics of the target architecture—like in this work. Furthermore, this technique is an important feature when one wishes to decrease the memory required to represent numbers and when exploiting certain advantages offered by the hardware and that depend on the representation of the data; i.e. faster integer operations available on many architectures.

4.3.7 Lookup Tables for Activation Functions

A lookup table is an array that stores pre-computed values for a given operation, replacing runtime computations with a simpler array indexing it. The savings in terms of processing time can be significant, since retrieving a value from memory is often faster

Figure 4.12: Scaling the weights and images

```
//#############################################################
// Scaling coefficients and defining integer size
//#############################################################
typedef int16_t t_weight;
typedef int16_t t_image;

#define IMAGE_UPPER_LIMIT 4.0   // Experimentally found limit
#define WCI_UPPER_LIMIT   4.0   // Experimentally found limit
#define WCI_LOWER_LIMIT  -4.0   // Experimentally found limit
#define W2_LOWER_LIMIT   -4.0   // Experimentally found limit
#define IMAGE_SCALE_COEFF 31.0  // = 127 / WCI_UPPER_LIMIT or IMAGE_UPPER_LIMIT
#define WEIGHTS_COEFF_WGI 31.0  // = 127 / WCI_UPPER_LIMIT or IMAGE_UPPER_LIMIT
#define WEIGHTS_COEFF_WGF 31.0  // = 127 / WCI_UPPER_LIMIT or IMAGE_UPPER_LIMIT
#define WEIGHTS_COEFF_WGO 31.0  // = 127 / WCI_UPPER_LIMIT or IMAGE_UPPER_LIMIT
#define WEIGHTS_COEFF_WCI 31.0  // = 127 / WCI_UPPER_LIMIT or IMAGE_UPPER_LIMIT
#define WEIGHTS_COEFF_W2  31.0  // = 127 / W2_LOWER_LIMIT  or IMAGE_UPPER_LIMIT
//#############################################################
...
// weight used by the neuron dot products of the hidden layer
WS = new t_weight [NUMBER_OF_HIDDEN_LAYERS * NUMBER_OF_GATES * NUMBER_OF_NEURONS * ↩
    NUMBER_OF_INPUTS];
...
// weight used by the softmax function in the output layer
W2 = new t_weight [NUMBER_OF_CLASSES * (1 + NUMBER_OF_NEURONS * 2)];
...
//vectors with the images in forward and backward directions
image_fw = new t_image [numberOfColumns * HIGHT_IN_PIX];
image_bw = new t_image [numberOfColumns * HIGHT_IN_PIX];
...
//#############################################################
// Scale the weights
//#############################################################
if(local_count < NUMBER_OF_NEURONS * NUMBER_OF_INPUTS_SSE)          // WGI
   WS[local_count] = WEIGHTS_COEFF_WGI * weight;
else if(local_count < 2 * NUMBER_OF_NEURONS * NUMBER_OF_INPUTS_SSE) // WGF
   WS[local_count] = WEIGHTS_COEFF_WGF * weight;
else if(local_count < 3 * NUMBER_OF_NEURONS * NUMBER_OF_INPUTS_SSE) // WGO
   WS[local_count] = WEIGHTS_COEFF_WGO * weight;
else                                                                // WCI
{
   //#######################################
   // Limit the range and scale the weight
   //#######################################
   if(weight > WCI_UPPER_LIMIT)
      weight = WCI_UPPER_LIMIT;

   if(weight < WCI_LOWER_LIMIT)
      weight = WCI_LOWER_LIMIT;

   WS[local_count] = WEIGHTS_COEFF_WCI * weight;
}
//#############################################################
...
//#############################################################
// Scale the input images
//#############################################################
if(pix > IMAGE_UPPER_LIMIT)
    pix = IMAGE_UPPER_LIMIT;
image_fw[column] = IMAGE_SCALE_COEFF * pix;
...
//#############################################################
```

than undergoing expensive computations or input/output operations. Each neuron activation function operation are pre-calculated and stored in static arrays (one array per operation) and “pre-fetched” as part of the program’s initialization phase (memoization).

This method implies in a small loss of accuracy since the results of the operations can be approximated according to the inputs.

All five activation functions of the LSTM memory cell can be implemented using 2*256 lookup tables (experimentally better runtime given the lost of accuracy). The initialization of the lookup tables is made with the reference functions from $< math.h >$ header—see Figure 4.13. The header and source files used to create the lookup tables in C++ language is presented in Appendix A in Figure A.1 and A.2, respectively. The lower- and upper-bound on the number entries for each function were experimentally obtained and passed as a parameter to the initialization of each operation since we wish the network to be accurate only for the range of inputs given to the operations used by the NN.

Figure 4.13: Initializing lookup table operations

```
1
2  //#####################################################################
3  // Structures required for LUTs
4  //#####################################################################
5  LUTs _hw[NUMBER_OF_FUNCTIONS_TOBE_APPROXIMATED];
6  // The vector of pointers to member-functions of LUTs class;
7  // The member-functions return float and receive float as an argument
8  std::vector<float(LUTs::*)(float)> _functions;
9
10 ...
11
12 //#####################################################################
13 // Initialization of the LUTs functions
14 //#####################################################################
15 unsigned int numberOfLUTs = 2 * 256;
16 //4 * 256: Accuracy: 98.2343\%
17 //2 * 256: Accuracy: 98.2267\%
18 //256: Accuracy: 98.1795\%
19
20 // Initial configuration of the vector with the functions
21 // Initialize it with the reference functions from <math.h>
22 _functions.push_back(&LUTs::LUT_coarse);//0
23 _functions.push_back(&LUTs::LUT_coarse);//1
24 _functions.push_back(&LUTs::LUT_coarse);//2
25 _functions.push_back(&LUTs::LUT_coarse);//3
26 _functions.push_back(&LUTs::LUT_coarse);//4
27 _functions.push_back(&LUTs::LUT_coarse);//5
28
29 _hw[0].Init("tanh_2"   , &tanhf   , -7.59431, 9.01091, 1.0);
30 _hw[1].Init("tanh_1"   , &tanhf   , -7.59431, 9.01091, 1.0);
31 _hw[2].Init("exp_1"    , &expf    , -23.8223, 23.2473, 1.0);
32 _hw[3].Init("sigmoid_2", &sigmoidf, -56.0446, 17.3286, 1.0);
33 _hw[4].Init("sigmoid_3", &sigmoidf, -66.8248, 17.3286, 1.0);
34 _hw[5].Init("sigmoid_1", &sigmoidf, -88.7229, 17.3286, 1.0);
35
36 _hw[0].GenerateLUTs(numberOfLUTs);
37 _hw[1].GenerateLUTs(numberOfLUTs);
38 _hw[2].GenerateLUTs(numberOfLUTs);
39 _hw[3].GenerateLUTs(numberOfLUTs);
40 _hw[4].GenerateLUTs(numberOfLUTs);
41 _hw[5].GenerateLUTs(numberOfLUTs);
42 //#####################################################################
```

Note that the software optimization techniques discussed so far are hardware-independent; all of them can be deployed on any target architectures and executed without

any further modifications. In the next section, we discuss hardware-dependent optimization techniques that can further improve the performance of a neural network.

4.4 Hardware Dependent Optimization Techniques

In this work we consider any optimization techniques that require modifications to be compiled/executed in different architectures as being *hardware-dependent*. This section discusses many possible hardware-dependent optimization techniques that can be deployed to speed up neural networks.

Single Instruction Multiple Data (SIMD) instructions, presented in Section 2.4, are fundamental building blocks of low-level parallelization and performance gains on CPUs. These instructions perform multiple operations in parallel on contiguous data, thereby making the issues of data locality discussed in Section 4.3.3 even more critical. Most modern Intel and ARM CPUs have SIMD instructions available, but it is important to note that they are architecture-dependent, since each architecture can be built to include them or not.

When using SIMD instructions, data layout; to achieve best performance, it is important to observe a few properties of these instructions. First, SIMD instructions generally operate faster on 16-byte blocks that are 16-byte aligned in memory, both if using SSE or NEON extensions. AVX extensions, on the other hand, work faster on 32-byte blocks that are 32-byte aligned in memory. Being 16- or 32-byte aligned implies that the memory address of the first byte of a memory block is a multiple of one of these values. As consequence, if an array of data to be processed via SIMD instructions is not aligned to 16 or 32 bytes, performance may be negatively affected. Forcing 16-byte and 32-byte alignment of a memory block can be done in C++ by replacing calls to $malloc()$ or new with calls to $posix\_memalign()$, or by using custom allocators; see Figure 4.14 for an example of alignment of the WS weight. Note, also, that since every SIMD instruction operates on a block of 16 or 32 bytes, if the size of a data vector is not a multiple of 16 or 32 bytes, computation will suffer from corner cases where the vector computation is affected to adapt data loaded to be processed without the complete word size. The solution to this problem is to include extra positions with zeros at the end of the data vector, as previously discussed in Section 4.3.3.

Figure 4.14: Example of 16-byte alignment of a memory block

```
1        // WS -> 16byte memory aligned
2        if((posix_memalign((void **)&WS, 16, sizeof(float) * 2 * 4 * ↩
             NUMBER_OF_NEURONS * NUMBER_OF_INPUTS_SSE)) != 0){
3            std::cerr << "posix_memalign WS";
4            exit(EXIT_FAILURE);
5        }
```

4.4.1 SSE Intrinsic Functions

SIMD instructions can be used by means of assembly instructions. Since these are not easy to program, thin wrapper functions known as *intrinsic functions* have been made available for the most popular C and C++ compilers which convert these wrapper functions into assembly instructions.

Streaming SIMD Extensions (SSE) is a hardware technology that enables single instruction multiple data on Intel and AMD CPUs (see Section 2.4.1 for more details). SSE instructions typically operate on 16-bytes chunks of data, which may correspond to 2 doubles, 4 floats, 8 shorts, or 16 bytes. Various data types have been defined by Intel on special headers to use the intrinsics (such as $< mmintrin.h >$ and $< pmmintrin.h >$ according to the technology version of SSE) to represent these types: $\_\_m128i$, $\_\_m128$ and $\_\_m128d$, representing, respectively, 8x16-bit integers, 4x32-bit floating-points, and 2x64-bit doubles. As an example of how to use SSE intrinsic functions, consider the case when one has two vectors of 4 floats: $sources = [s1, s2, s3, s4]$ and $weights = [w1, w2, w3, w4]$. We can obtain the resulting sum vector, $addition = [s1 + w1, s2 + w2, s3 + w3, s4 + w4]$, by writing (in C or C++) the code shown in Figure 4.15 with just one instruction using the $\_\_m128$ datatype.

Figure 4.15: Vector addition example with SSE intrinsic function

```
1 #include <mmintrin.h>
2 __m128 c = _mm_add_ps(a, b);
```

In order for a compiler to be able to translate the wrapper functions specified by intrinsics to assembly instructions, particular compilation flags are needed. These can be enabled according to the specific SSE version one wishes to use (e.g. -msse2, -mssse3, -msse4, -msse4.1, -msse4.2) or based on the target architecture where the software will be compiled in (e.g. -march=native).

Intel processors that support SSE2 (one of the first SSE versions available on Intel architectures) provide the basic instructions to perform the multiply-and-add steps using floating-point SIMD arithmetic via the following functions: $\_mm\_add\_ps()$ and

$\_mm\_mul\_ps()$. Figure 4.16 shows a simple example where one wishes to accumulate the dot product of $\_\_m128 * sources$ and $\_\_m128 * weights$ to a third vector, $\_\_m128\, sum$.

Figure 4.16: Dot product with SSE2 intrinsic functions

```
1 //mul[0] = sources[0]*weights[0], ..., mul[3] = sources[3]*weights[3]
2 __m128 mul = _mm_mul_ps(*sources, *weights);
3 //sum[0] += mul[0], ..., sum[3] += mul[3]
4 sum = _mm_add_ps(mul, sum);
```

In the example depicted in Figure 4.16, the variable sum stores 4 partial sums which have to be added horizontally (addition of all partial sums) to yield the final result via a mechanism such as horizontal sums ($\_mm\_hadd\_ps()$) available on SSE3. Most modern compilers are capable of leveraging SSE or other kinds of SIMD instructions automatically from simple codes. However, automatic vectorization does not come close to the performance gains that one can achieve by writing code that leverages such instructions explicitly.

In the previous sections we have introduced two general ways in which a neural network can be sped up, by trading-off speed and accuracy: one where weights and inputs are represented with floating point numbers, and another one where this data is stored as short integers. With that in mind, we now highlight the main SSE intrinsic functions that can be used in each one of these cases in order to perform a dot product; more details of each one of these intrinsics can be found in (INTEL..., 2011):

1. definition of the vectors using SSE data types:

```
FLOAT  : __m128 s1;
INTEGER: __m128i s1;
```

2. initialization of accumulators:

```
FLOAT  : sum = _mm_set1_ps(0.0f);
INTEGER: sum = _mm_setzero_si128();
```

3. conversion from normal vectors to SSE data types:

```
FLOAT  : __m128  *pSource = (__m128*)source;
INTEGER: __m128i *pSource = (__m128i*)source;
```

4. main dot product computation via multiply and accumulate wrappers:

```
FLOAT  : s1 = _mm_add_ps(s1,_mm_mul_ps(*pSource,*pWeight));
INTEGER: s1 = _mm_add_epi32(s1,_mm_madd_epi16(*pSource,*pWeight));
```

5. performing horizontal sums to store the result in the (standard) vector received as parameter:

```
FLOAT  : s1234 = _mm_hadd_ps(_mm_hadd_ps(s1,s2),_mm_hadd_ps(s3,s4));
INTEGER: s1234 = _mm_hadd_epi32(_mm_hadd_epi32(s1,s2),_mm_hadd_epi32(s3,s4));
```

Since dot product functions that manipulate integers are more complex and have more details than it would be feasible to show in this document, we introduce in Figure 4.17 a simplified version of an integer implementation of dot product (required by neurons) which makes use of SSE intrinsics. The float implementation of the dot product is similar, except for the use of different intrinsics, as explained above, and also in jumps over data vectors that are made after every 128 bits (4 positions for floats in contrast of 8 positions for short integers). Since different weights in an optimized neural network are stored in a same vector, in order to make better use of cache locality, some fixed jumps can be seen in the first parameter of $\_mm\_madd\_epi16$ wrapper in Figure 4.17. When designing an optimized integer implementation of the dot product, we performed many experiments to analyze the best memory layout and access patterns to set this. The above mentioned optimized integer implementation of the dot product, using SSE intrinsic functions, can be used to accelerate both the computations in the hidden layer of a BLSTM, as well as the dot products required by the softmax in the output layer. Experimental analyses of these optimization techniques are presented in Chapter 6.

4.4.2 AVX Intrinsic Functions

Advanced Vector Extensions (AVX) are more recent extensions to the x86 instruction set architecture for microprocessors from Intel (see Section 2.4.2 for more details). AVX extends the size of the registers that perform SIMD and its instructions so that they typically operate on 32-bytes chunks of data at a time. By using these extensions, it is possible to optimize neural network operations so that an implementation using AVX can process 8 floats and 16 integers at a time with just one instruction. Various intrinsic functions and data types are provided with this extension set. The data types that we use when evaluating optimized neural networks that use AVX are $\_\_m256i$ and $\_\_m256$, which represent, respectively, 16 16-bit integers and 8 32-bit floating points.

We present, below, the main AVX intrinsic functions that can be used in integer and floating point implementations of the dot product operations required by a BLSTM NN:

1. definition of the vectors using AVX data types:

```
FLOAT  : __m256
INTEGER: __m256i
```

2. initialization of accumulators:

Figure 4.17: Integer implementation of dot product function with SSE intrinsics

```
inline void DotProduct128_SSE(t_weight *source,
                              t_weight *WS,
                              int32_t  *output){

    // DEFINING AND INITIALIZING ACCUMULATORS
    __m128i s1 = _mm_setzero_si128();
    __m128i s2 = _mm_setzero_si128();
    __m128i s3 = _mm_setzero_si128();
    __m128i s4 = _mm_setzero_si128();

    // CONVERTING NORMAL VECTORS TO SSE DATATYPES
    __m128i *pWS     = (__m128i*)WS;
    __m128i *pSource = (__m128i*)source;
    __m128i *pOutput = (__m128i*)output;

    for (unsigned int i = 0; i < NUMBER_OF_INPUTS_SSE/8; i++)
    {
        // MULTIPLY AND ACCUMULATE (dot product)
        s1 = _mm_add_epi32(s1, _mm_madd_epi16(*(pWS + (0 * NUMBER_OF_INPUTS_SSE*↩
            NUMBER_OF_NEURONS/8)), *pSource));
        s2 = _mm_add_epi32(s2, _mm_madd_epi16(*(pWS + (1 * NUMBER_OF_INPUTS_SSE*↩
            NUMBER_OF_NEURONS/8)), *pSource));
        s3 = _mm_add_epi32(s3, _mm_madd_epi16(*(pWS + (2 * NUMBER_OF_INPUTS_SSE*↩
            NUMBER_OF_NEURONS/8)), *pSource));
        s4 = _mm_add_epi32(s4, _mm_madd_epi16(*(pWS + (3 * NUMBER_OF_INPUTS_SSE*↩
            NUMBER_OF_NEURONS/8)), *pSource));

        // JUMPS TO NEXT DATATYPE POSITION (8 positions to integers)
        pSource++;
        pWS++;

    }

    // HORIZONTAL SUMS TO RETURN RESULT TO NORMAL VECTOR
    __m128i s1234 = _mm_hadd_epi32(_mm_hadd_epi32(s1,s2),_mm_hadd_epi32(s3,s4));

    // NORMAL VECTOR (output) RECEIVES 4 POSITIONS THROUGH (pOutput)
    pOutput[0] = s1234;

}
```

We use the header $\#include < pmmintrin.h >$ and $-march = native$ compiler flag to make use of the intrinsics for this software configuration.

```
FLOAT   : sum = _mm256_setzero_ps();
INTEGER: sum = _mm256_setzero_si256();
```

3. conversion from normal vectors to AVX data types:

```
FLOAT   : (__m256  *pSource = (__m256*)source;)
INTEGER: (__m256i *pSource = (__m256i*)source;)
```

4. main dot product computation via multiply and accumulate wrappers:

```
FLOAT   : (s1 = _mm256_fmadd_ps(*pSource,*pWeight,s1);)
INTEGER: (s1 = _mm256_add_epi32(s1, _mm256_madd_epi16(*pSource,*pWeight));)
```

5. horizontal sums and conversions from AVX data types to vector data types, in order to store the results in the (normal) vector received as parameter:

```
FLOAT   :  (__m256 s1234 = _mm256_hadd_ps(_mm256_hadd_ps(s1,s2),_mm256_hadd_ps(s3,s4));)
           (__m128 low1234  = _mm256_extractf128_ps (s1234, 0);)
           (__m128 high1234 = _mm256_extractf128_ps (s1234, 1);)
           (__m128 out = _mm_add_ps(low1234, high1234);)
INTEGER:  (__m256i s1234 =  _mm256_hadd_epi32(_mm256_hadd_epi32(s1,s2), _mm256_hadd_epi32
```

```
(s3,s4));)
          (__m128i low1234  = _mm256_extractf128_si256(s1234, 0);)
          (__m128i high1234 = _mm256_extractf128_si256(s1234, 1);)
          (__m128i out = _mm_add_epi32(low1234, high1234);)
```

Figure 4.18 presents a simplified version of an AVX-based integer implementation of the dot product required to compute the output of neurons. The AVX float implementation is similar except for the use of different intrinsics, as previously explained, and in jumps over data vectors that are made after every 256 bits (8 positions for floats in contrast of 16 positions for integers).

Figure 4.18: Integer implementation of dot product function with AVX intrinsics

```
inline void DotProduct128_AVX(t_weight *source_fw,
                              t_weight *WS,
                              int32_t *output){

    // DEFINING AND INITIALIZING ACCUMULATORS
    __m256i s1 = _mm256_setzero_si256();
    __m256i s2 = _mm256_setzero_si256();
    __m256i s3 = _mm256_setzero_si256();
    __m256i s4 = _mm256_setzero_si256();

    // CONVERTING NORMAL VECTORS TO AVX DATATYPES AND OUTPUT TO SSE DATATYPE FOR CONVERSION PURPOSE
    __m256i *pWS     = (__m256i*)WS;
    __m256i *pSource = (__m256i*)source;
    __m128i *pOutput = (__m128i*)output;

    for (unsigned int i = 0; i < NUMBER_OF_INPUTS_SSE/16; i++)
    {

        // MULTIPLY AND ACCUMULATE (dot product)
        s1 = _mm256_add_epi32(s1, _mm256_madd_epi16(*(pWS + (0 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS/16)), *pSource));
        s2 = _mm256_add_epi32(s2, _mm256_madd_epi16(*(pWS + (1 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS/16)), *pSource));
        s3 = _mm256_add_epi32(s3, _mm256_madd_epi16(*(pWS + (2 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS/16)), *pSource));
        s4 = _mm256_add_epi32(s4, _mm256_madd_epi16(*(pWS + (3 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS/16)), *pSource));

        // JUMPS TO NEXT DATATYPE POSITION (16 positions to integers)
        pSource++;
        pWS++;

    }

    // HORIZONTAL SUMS AND DATATYPE CONVERSIONS TO RETURN RESULTS TO NORMAL VECTOR
    __m256i s1234 = _mm256_hadd_epi32(_mm256_hadd_epi32(s1,s2),_mm256_hadd_epi32(s3,s4));
    __m128i low1234  = _mm256_extractf128_si256(s1234, 0);
    __m128i high1234 = _mm256_extractf128_si256(s1234, 1);
    __m128i out = _mm_add_epi32(low1234,high1234);

    // NORMAL VECTOR (output) RECEIVES 4 POSITIONS THROUGH (pOutput)
    pOutput[0] = out;

}
```

To have access to the intrinsic functions provided by Intel we use the header $\#include < pmmintrin.h >$ and $-march = native$ compiler flag. It could also be enabled by the compiler flags $-mavx$ and $-mavx2$ according to the intrinsic functions used.

Finally, we highlight that it is possible to use AVX intrinsic functions as an additional way to speed up the hidden layer and softmax dot products in an optimized NN. In subsequent experiments and analyses, we will compare a network optimized with AVX intrinsics with ones optimized using features available only on Intel architectures.

4.4.3 NEON Intrinsic Functions

NEON technology is the ARM architecture SIMD extension (see Section 2.4.3 for more details). The use of NEON intrinsics is similar to the use of SSE: the data structures used with NEON intrinsics must, for example, be aligned in memory blocks of at least 16 bytes. Figure 4.19 shows an integer implementation of the dot product via NEON intrinsics.

The code depicted in Figure 4.19 is organized in a similar way to the dot product implementation using SSE and AVX intrinsics. We first initialize the accumulators with zeros and create special vectors to the NEON data type registers for each one of our normal vectors received as parameters. Inside the main loop, we load the normal vectors to the special vectors used by NEON. After this, multiplications and additions are performed with the use of two intrinsic functions: *vaddq_s32()* and *vmulq_s16()*. Since the NEON intrinsic *vmulq_s16()* returns an *int16x8_t* data type, and we need to perform the addition of two *int32x4_t* vectors via the *vaddq_s32()* function, we apply the function *_convert_to_32x4int(int16x8_t a, int16x8_t b)* to convert the multiplication results of *vmulq_s16()* from integers of 16 bits to integers of 32 bits before executing the dot product accumulation. At the end of the procedure, we use another conversion function to store the values currently in the NEON data type (i.e. accumulators: s1, s2, s3 and s4) into the normal vector pointed by **weights* parameter. The floating point implementation of this procedure is similar, except that it uses just one function to perform the vector multiplications and accumulations: *vmlaq_f32(float32x4_t a, float32x4_t b, float32x4_t c);*. Such a function implements both multiplication and addition simultaneously with a same underlying assembly instruction.

Figure 4.19: Integer implementation of dot product function with NEON intrinsics

```
 inline __attribute__((always_inline)) int32_t _convert_to_int(int32x4_t a )
{
   return vget_lane_s32(vpadd_s32(vpadd_s32(vget_high_s32(a), vget_low_s32 (a)), vpadd_s32(vget_high_s32(a), vget_low_s32 (a))),0);
}

inline __attribute__((always_inline)) int32x4_t _convert_to_32x4int(int16x8_t a, int16x8_t b)
{
   return vpaddlq_s16( vmulq_s16( a , b ) ); // load vectors of 16 x 8 bits ints multiply and add
}

inline void DotProduct128_NEON(t_weight *source,
                               t_weight *WS,
                               int32_t *weights)
 {

     // DEFINING AND INITIALIZING ACCUMULATORS
     int32x4_t s1 = {0, 0, 0, 0}, s2 = {0, 0, 0, 0}, s3 = {0, 0, 0, 0}, s4 = {0, 0, 0, 0};

     // DEFINING NEON DATATYPE REGISTERS TO LOAD NORMAL VECTORS TO NEON REGISTERS
     int16x8_t pSource,pWS1,pWS2,pWS3,pWS4;

     for (unsigned int i = 0; i < NUMBER_OF_INPUTS_SSE; i+=8)
     {
         // LOADING SOURCE AND WEIGHTS TO MEMORY
         pSource=vld1q_s16( &source[i] );
         pWS1=vld1q_s16( &WS[i+(0 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS)] );
         pWS2=vld1q_s16( &WS[i+(1 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS)] );
         pWS3=vld1q_s16( &WS[i+(2 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS)] );
         pWS4=vld1q_s16( &WS[i+(3 * NUMBER_OF_INPUTS_SSE*NUMBER_OF_NEURONS)] );
         // MULTIPLY AND ACCUMULATE (dot product)
         s1 = vaddq_s32( s1, _convert_to_32x4int(pWS1, pSource) );
         s2 = vaddq_s32( s2, _convert_to_32x4int(pWS2, pSource) );
         s3 = vaddq_s32( s3, _convert_to_32x4int(pWS3, pSource) );
         s4 = vaddq_s32( s4, _convert_to_32x4int(pWS4, pSource) );

     }

     // RETURNING RESULTS TO NORMAL VECTOR
     weights[0] = _convert_to_int(s1);
     weights[1] = _convert_to_int(s2);
     weights[2] = _convert_to_int(s3);
     weights[3] = _convert_to_int(s4);

 }
```

The built-in intrinsics for the ARM Advanced SIMD extension are available when the $-mfpu = neon$ compiler flag is used. The NEON intrinsics are defined in the header file $arm\_neon.h$. The header file defines both the intrinsics and set of vector types.

5 EXPERIMENTAL METHODOLOGY

This chapter describes how we will perform experiments in order to evaluate the performance and accuracy of the different optimization strategies we described in the previous chapters. We also discuss the metrics that will be used to measure the possible improvements obtained by deploying them. Specific results and numerical analyses will be presented in Chapter 6. The main metric used to evaluate the accuracy of our proposed optimized BLSTM implementation is, as discussed in Section 2.3.1, the Levenshtein Distance. In this chapter, we also introduce how differently optimized neural networks are evaluated and compared with respect to runtime and energy consumption metrics.

5.1 Evaluation Problem

As discussed in Section 2.2, in this work we choose to evaluate our optimized BLSTM in an Optical Character Recognition problem. This decision was made because OCR is a typical sequence recognition task and in this problem, BLSTMs have shown superior performance in terms of accuracy and robustness, when compared to other types of neural networks. Furthermore, OCR is an important component part used by many different different real-world applications, such as cognitive computing, machine translation, text-to-speech, and text mining. Next, we present the ways in which we evaluate and compare the different implementation configurations of our optimized neural networks.

5.2 Comparisons

In order to evaluate the advantages gained by combining the proposed software optimization techniques and by exploiting specific hardware capabilities, we evaluate our differently-optimized networks under different conditions. Concretely, we measure different performance metrics, such as accuracy, runtime, and energy consumption, when running each particular implementation in a different hardware architecture, and when deploying specific combinations of the proposed optimization techniques—for instance, when combining precision reduction, intrinsic functions, lookup tables, etc. In Chapter 6 we will discuss the results obtained by executing such comparisons.

5.3 Evaluation Metrics

Each possible configuration of an optimized neural network will, as mentioned in Section 5.2, be evaluated according to three main metrics, which we discuss below.

5.3.1 Accuracy

As discussed in Section 2.3, one way to evaluate the accuracy of an NN is by comparing its outputs with respect to true reference, or ground truth outputs. This metric is especially important when implementing optimizations that can affect the accuracy of the intermediate numeric results calculated by the network, such as when using optimizations that involve numerical precision reduction. To measure accuracy, we use a testing set containing T images as inputs and T ground truth strings; the latter are compared to each correspondingly produced NN output. Since we are, in this work, evaluating the network on as OCR problem, the network output is a string. To compare the similarity between the NN predicted string and the testing set ground truth, we use a function that compares all of the resulting strings processed by the neural network with their corresponding ground truth outputs. The overall BLSTM neural network accuracy is then defined as the mean distance between between all of the network output strings and their associated testing set ground truth strings.

The recognition accuracy of the un-optimized baseline BLSTM of is 98.2337%, when using single-precision floating-point format to store data and weights. In Chapter 6 we show that runtime can be improved by applying the software optimization techniques presented here, and exploiting hardware capabilities, without significantly decreasing this reference recognition accuracy.

5.3.2 Runtime

In this work, runtime (or execution time) is defined as the processing time required to process the instructions related to the output computation of the BLSTM neural network. The runtime of the NN is measured in milliseconds in order to better control the impact of our improvements. Note that runtime, as defined above, is the execution or deployment time of the NN with respect to applying it to predict the output to novel inputs;

it does not include the time needed to train the network, since optimizing it is not the main goal of this work.

As discussed in Section 4.1, we begin measuring the neural network runtime immediately after reading all input images and data structures to memory. When the neural network finishes its processing, by storing all recognized strings to memory, we collect the final processing time stamp. By comparing that time stamp with the one when runtime began being measured, we obtain the (wall-clock) time required to process all testing instances. In this work we perform time measurements by using a time library for the C++ language, called <*chrono*>.

Each modification to the baseline un-optimized BLSTM will have its runtime measured to later comparison with other possible software configurations.

5.3.3 Energy Consumption

Energy consumption is an important metric for embedded systems in which the energy budget is restricted. Given that the mobile market is one of the most important markets in terms of devices where Artificial Intelligence can be deployed, we also, in this work, evaluate our proposed software optimizations with respect to the resulting energy consumption required to execute a given neural network.

To calculate the energy consumption of a given optimization configuration of a BLSTM NN we first estimate the idle power of a given target architecture, by performing direct measurements when possible (e.g. on a laptop processor, Raspberry pi 3 Model B Board, and Zynq board) or by checking its documentation (e.g. XEON). We then make power measurements while the neural network is being executed, by using a multimeter placed between the component that uses CPU power and the power source. The multimeter is connected to another computer while it collects measurements, resulting in a power consumption sample for each processing second of the NN. At the end of the processing of the NN, we make the total power estimation by calculating the mean of all power measurements obtained minus the idle power estimated for the corresponding architecture. The NN processing power is multiplied by the NN runtime, resulting in the NN energy consumption estimation, since Energy $=$ Power $\times$ Time. We use a multimeter in all experiments that require direct power measurement, except for measuring it on laptops; in this case the measurements are made by using the Powerstat software.

To summarize, four main steps are involved in estimating the energy consumption

required to execution a given optimized or un-optimized BLSTM: *1)* direct measurement or estimation of the architecture power while in idle P_{idle}; *2)* execution of the NN and measurement of the mean architecture power dissipation during execution P_{exec}; *3)* calculation of the power required just by the NN processing P_{BLSTM}; that is $P_{BLSTM} = P_{exec} - P_{idle}$; *4)* estimation of the energy consumption, performed by multiplying P_{BLSTM} and the NN runtime: $Energy_{BLSTM} = P_{BLSTM} \times Runtime_{BLSTM}$.

5.4 Compilation Settings for Experiments

The experiments described in Chapter 6 are influenced by the way the software implemented, using different optimization techniques subsets, is compiled. Our implementations of the optimization techniques are compiled with gcc and icc compilers, depending on the particular target architecture being considered.

The gcc compiler is used with -O3 and -std=c++11 flags for the first configurations of our software (baseline, just with online and offline parallelization and with the loop unrolling technique) and with the -Ofast flag, instead -O3, for all other optimizations. We used the icc compiler when testing our implementations on the supercomputer architecture (Xeon), and it presented better results compared to gcc. We always compiled directly on the target architecture, since our attempts to cross-compilation resulted in worse runtime. Some additional optimization flags were used in both compilers in order to enable specific hardware features, as discussed in Section 4.4.

6 RESULTS

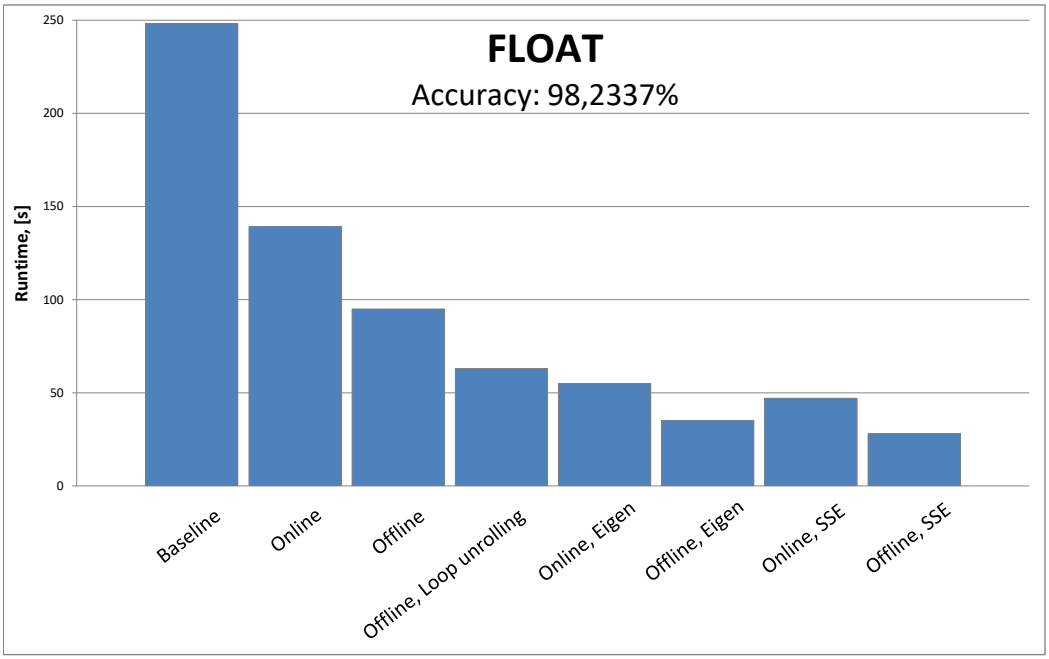
This chapter presents the performance evaluation of the neural networks resulting from the use of the proposed optimization techniques. Some of the software optimization techniques discussed in Section 4 were analyzed in (RYBALKIN et al., 2017); in this chapter, we present novel results which were not presented in that article, including the use of different combinations of software-based optimizations, and of optimization strategies that exploit particular features of a given target architecture. We also present a more in-depth discussion and analysis of each result obtained by deploying different optimizations.

6.1 BLSTM runtime on Intel architectures

The first performance evaluation we conduct in this work consists in processing the testing set of images through the baseline floating-point NN implementation on the Intel i7-4500U architecture. We apply different optimization techniques to verify their effects and advantages. Each optimization technique is deployed individually or in combination with other optimizations, as will be discussed in more details in what follows. We collect results (performance measurements, as discussed in Section 5) and use them to compare the different optimization configurations that can be applied over the baseline un-optimized BLSTM. Once we identify a given optimization strategy that has a positive impact we re-apply it in combination with subsequent optimization techniques that we wish to verify.

Figure 6.1 shows the impact of different optimization techniques when applied to the floating-point implementation of the NN on an Intel architecture, without deployment of any techniques that may reduce network accuracy. The first improvement shown in this figure is the application of parallelization achieved via the OpenMP library, by implementing the online and offline scenarios introduced in Section 4.3.1. The performance data gathered for this optimization configuration suggests large runtime improvements, in the range of 1.7 times faster in the online parallelization scenario, and 2.6 times faster in the offline parallelization scenario. Since parallelizing different components of the network offers a clear runtime improvement, in subsequent experiments we will combine it with other optimization techniques, instead of evaluating them in isolation, when deployed to the baseline un-optimized network.

Figure 6.1: Runtime of a floating-point BLSTM implementation under different optimization techniques on the Intel i7-4500U @ 3.0 (turbo) architecture



Besides parallelization, another significant runtime reduction can be achieved (as shown in the fourth bar of Figure 6.1) by applying three new optimization techniques: *1)* aggregation of the hidden layers; *2)* contiguous memory allocation of weights and inputs; and *3)* loop unrolling of the dot product operations. Before deploying these improvements, computing the BLSTM output required processing two separated hidden layers with 4 different weights for each LSTM cell, resulting in the processing of 8 different weight arrays and dot product functions. The hidden layer aggregation strategy allows us to allocate just one array with all the data needed to process the NN output. With this single large weight array, it becomes easier to perform tests to determine the best memory allocation and access to the data, since each modification in this big array represents a big difference of memory usage by the architecture memories.

We evaluated the impact of deploying loop unrolling together with the above mentioned optimizations of hidden layers aggregation, new memory allocations, and new memory accesses in combination with the offline parallelization strategy. This resulted in an improvement of 1.5 times better runtime when compared to the offline parallelization scenario without any other optimization techniques. Since the dot product could be processed using a better approach (i.e. Eigen library and SSE instructions) we just show in Figure 6.1 the result obtained with the offline parallelization scenario. In what follows, we show that other optimization techniques can also contribute to speeding up the dot

product implementation.

In the subsequent experiments presented in this section, we use hidden layer aggregation and apply the idea of allocating all memory structures in a contiguous way. We then evaluate the effect of composing these base optimization strategies with others, in order to speed up the dot product operations, for instance, using the Eigen library (see more details in Section 4.3.5). The use of the Eigen library, in combination with the previously mentioned optimizations, results in an runtime improvement of 1.8 times, compared to the offline implementation that uses loop unrolling.

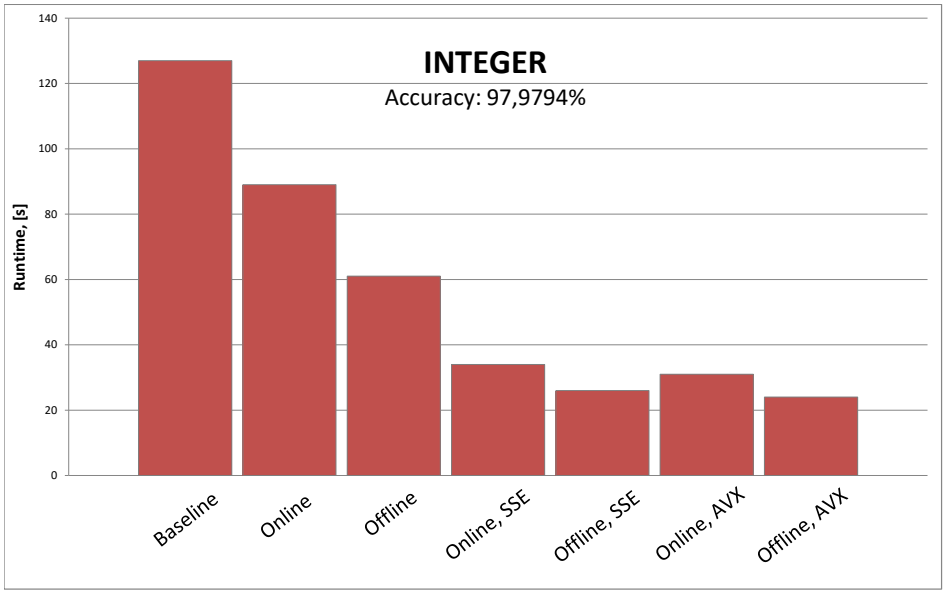
Since, in these experiments we are using an Intel architecture, we now consider the impact of using the SIMD instructions available on it. The use of SSE (discussed in Section 4.4.1) to accelerate the dot product operations results in a considerable runtime improvement, even compared to the dot product version using the highly optimized Eigen library. Our SSE implementation outperforms Eigen by 15% (when using online parallelization) by 20% (when using offline parallelization). It is important to mention that Eigen is a much more general library than just one to compute dot products, and the results we obtain here should not be construed as a general statement about the capabilities or performance of Eigen library in other applications.

Before presenting the next experimental results, one important thing to mention is the way in which we have compiled the different optimized NNs. While performing experiments, we realized that -Ofast flag provides a considerable runtime improvement, mainly if used to compile the less optimized versions of the network (baseline and with just online and offline parallelization). This happens mainly because this compilation flag optimizes loops, and the simplest optimization configurations that we evaluate here do not implement manually-constructed loop unrolls or exploit any kind of loop optimization. The loop-unrolled version of our baseline NN does not work properly (i.e. network accuracy lost) using -Ofast, demonstrating that this flag compiles all loops using loop optimizations. The first four bars in Figure 6.1 show the performance results using the -O3 compilation flag, while the remaining software configurations make use of -Ofast compilation flag. The same approach will be considered to the next results.

Figure 6.2 shows the effect of different optimization techniques when applied to the integer implementation of the NN on an Intel i7-4500U architecture. In this configuration of the code we convert all floating-point parameters and inputs to integer (see Section 4.3.6 for more details). This conversion results in a small accuracy reduction due to reduced numerical precision. We show in Figure 6.2 only the main combinations of

optimization techniques obtained in Figure 6.1 and make use of the architecture-specific AVX instruction set. Making use of AVX instructions decreases runtime on average by 9%, when compared to the use of SSE.

Figure 6.2: Runtime of an integer BLSTM implementation under different optimization techniques on the Intel i7-4500U @ 3.0 (turbo) architecture

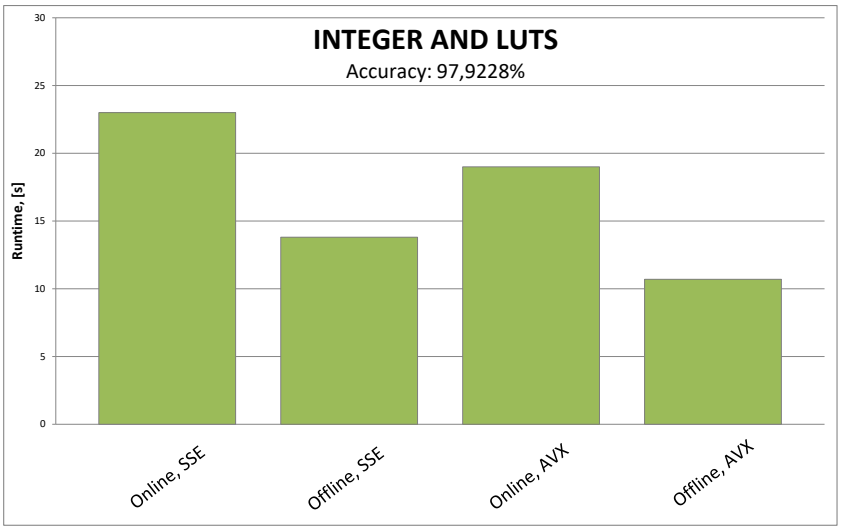


The trade-off between accuracy and runtime, which results from approximating floating point numbers with integers, is an important question to be discussed. We observed, in our experiments, that a small accuracy reduction (of only 0.25%) when using integers is balanced off by a large reduction in the data processed by the NN (each network parameter is represented with 16-bit per integer instead of 32-bit per float) and consequently a big reduction of runtime. Our experiments show that the integer-based NN optimized configuration of the NN can be executed at almost half of the runtime of the floating-point implementations. The SSE version of the integer network results in a runtime reduction of 34% (if using online parallelization) and 9% (if using offline parallelization) compared to the floating-point implementation of the network.

The last improvement proposed for the Intel i7-4500U architecture, which reduces accuracy in an almost insignificant way but that offers a large gain in runtime, is the use of lookup tables to approximate the neuron activation functions. The results obtained by evaluating the use of lookup tables (in an integer-based NN), and combining it with different parallelization scenarios, is presented in Figure 6.3.

Figure 6.4 aggregates all different optimized configurations of the NN when executed on an Intel i7-4500U. This figure clarifies which specific combination of optimization techniques allows for the best way of tackling the different computational bottlenecks

Figure 6.3: Runtime of an integer BLSTM implementation using lookup tables on the Intel i7-4500U @ 3.0 (turbo) architecture



of the baseline BLSTM. The best runtime we achieved when using a floating-point network is of 10.7 seconds, compared to the 248 seconds required for running the baseline implementation of the BLSTM. This corresponds to an improvement of 23.2 times. If applying the offline parallelization strategy to the integer implementation of the NN, along with better memory allocation, AVX instructions and lookup tables, runtime improves by approximately 9 times compared to the floating-point implementation with offline parallelization.

Figure 6.4: Complete runtime comparison of floating-point and integer BLSTM implementations on an Intel i7-4500U @ 3.0 (turbo) architecture

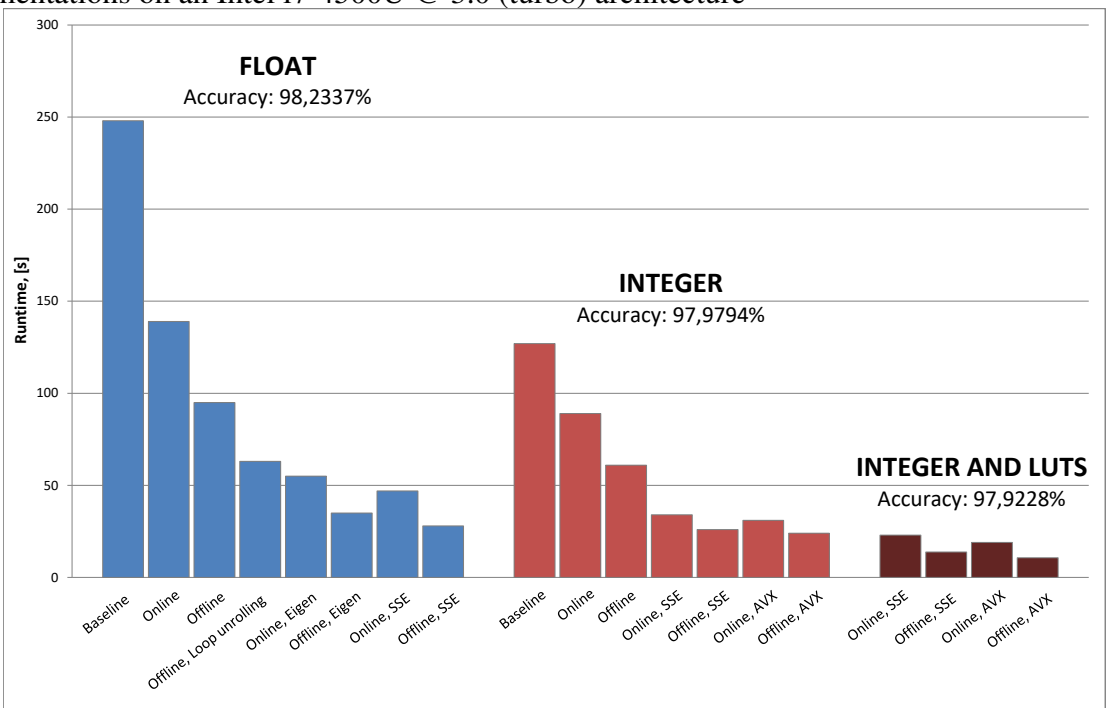
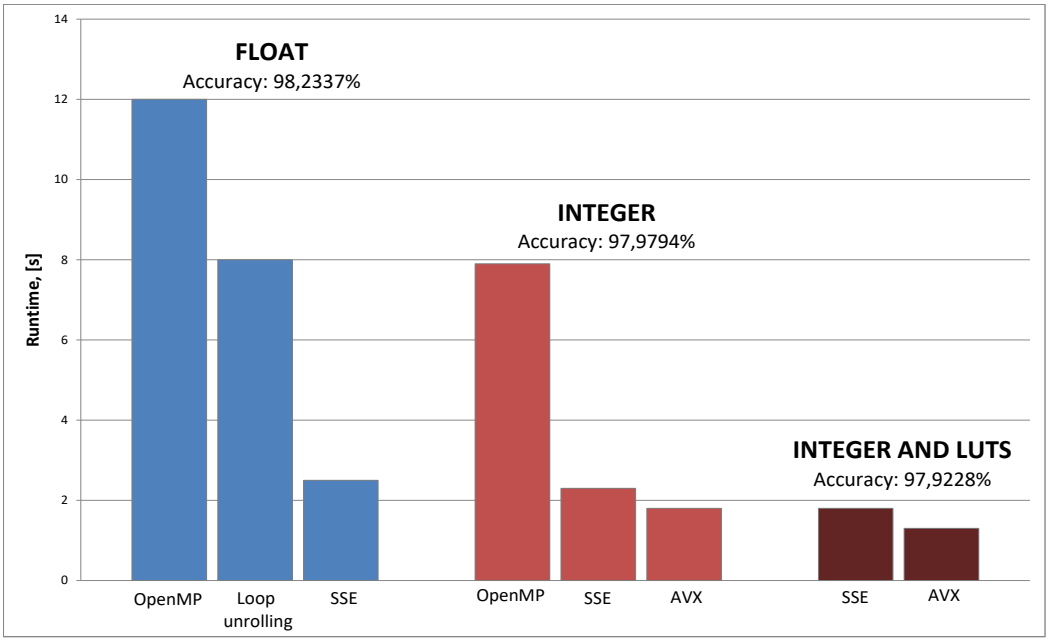


Figure 6.5 shows performance results for the offline parallelization scenario using different optimization techniques on a Xeon E5-2670 v3 architecture. We omit the results for online parallelization since they did not result in any significant runtime improvement. This occurred, we believed, due to the high inter-thread synchronization costs involved in fine-grained parallelization, which were exacerbated by the recurrent nature of the NN. If using offline parallelization, we observed a 9 times runtime improvement when comparing the floating-point software just with parallelization to the best results achieved via an integer-based NN with AVX instructions and lookup tables. If we compare the baseline floating-point implementation (omitted in Figure 6.5) with the best result achieved, this improvement is of up to 169 times (baseline with no parallelization runs in 220 seconds on Xeon).

Figure 6.5: Runtime comparison of floating-point and integer-based BLSTM implementations under the main optimization techniques, using just offline parallelization on a Xeon E5-2670 v3 @ 3.1 GHz (turbo) architecture



6.2 BLSTM runtime on ARM architectures

This section presents the runtime improvements achieved by the best optimized configurations of the NN when deployed on ARM architectures. We apply the same optimization techniques previously presented, with one key difference: the intrinsic functions that are used to implement dot products. Instead of using SSE or AVX (available on Intel architectures), we use the intrinsics provided by ARM architectures: NEON (see Section

4.4.3 for more details).

Figure 6.6 shows, on the left, the runtimes achieved by a floating point implementation of the BLSTM on an ARM Cortex-A53 (Raspberry Pi 3 board). The light blue bars show, respectively, the performance of the un-optimized baseline BLSTM and best optimization configurations we identified—both deploying different types of parallelization, using new memory blocks allocation, hidden layers aggregation, and NEON intrinsics. These floating point-based NNs did not incur any accuracy loss. The dark blue bars show the optimized floating-point configurations when applying lookup tables to approximate the activation functions, where a small reduction in accuracy is tolerated in exchange for better runtimes. The same type of performance evaluation shown for floating point NNs are repeated on an integer-based implementation of the BLSTM (light- and dark-red bars in Figure 6.6).

Figure 6.6: Runtime of floating-point and integer BLSTM implementations under the main configurations of software optimization techniques on an ARM Cortex-A53 @ 1.2 GHz architecture

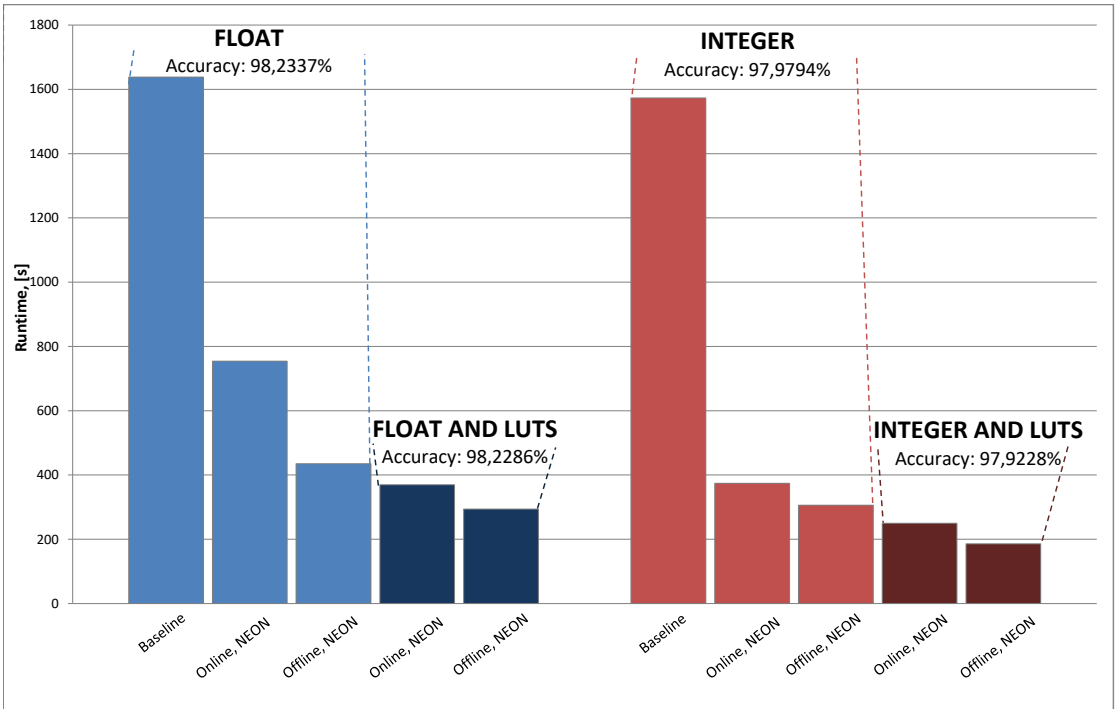
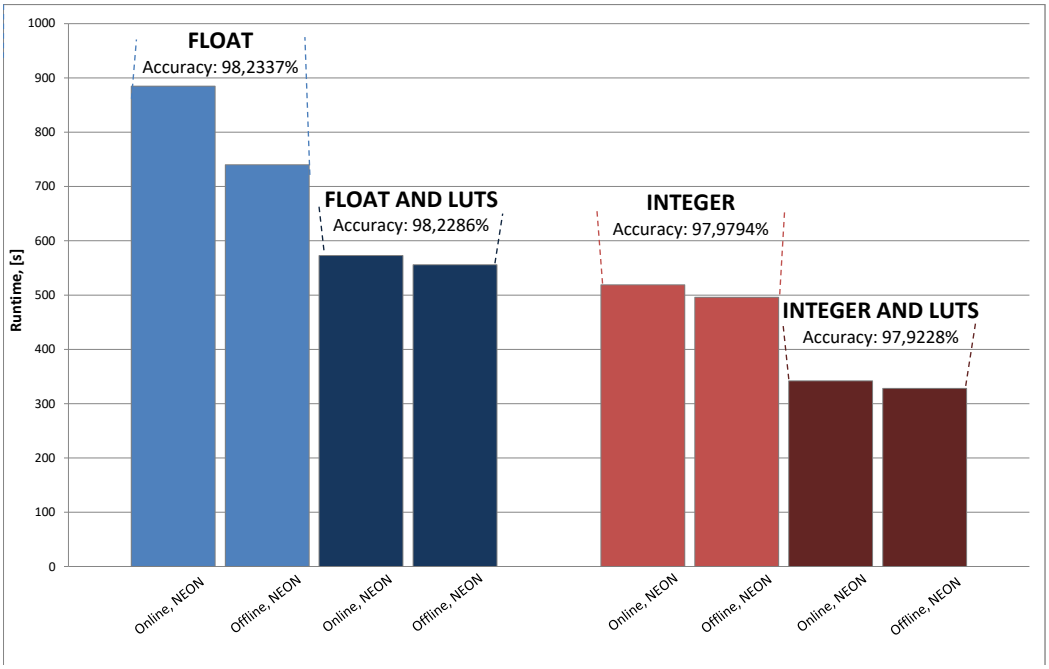


Figure 6.7 shows the comparison of the optimized floating-point implementations of the NN with the integer-based ones on an ARM Cortex-A9 (Zynq board). We omit the results of the baseline network due to their large runtimes, when compared to the results obtained via the optimized configurations presented here; in particular, the floating-point baseline implementation runs in almost 3000 seconds, while even the less efficient opti-

mized NN presented in Figure 6.7 runs in less than 900 seconds.

Figure 6.7: Runtime of floating-point and integer BLSTM implementations under the main configurations of software optimization techniques on an ARM Cortex-A9 @ 800MHz



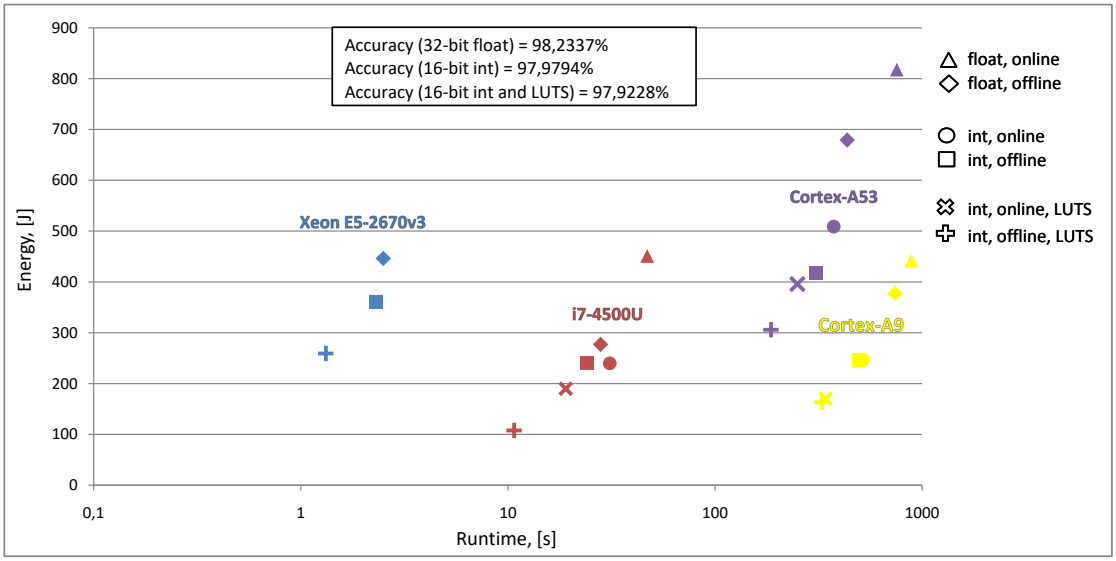
The results obtained on ARM architectures demonstrate, once again, that combining different subsets of the optimization techniques proposed in this work can result in large runtime improvements. The proposed optimizations for parallelization, memory allocation, memory aggregation, and use of intrinsics, results in the largest runtime reductions. In the cases where even better runtimes are required, the use of numerical precision reduction (by converting floats to integers) and using lookup tables, are valuable options.

6.3 Runtime and energy consumption on Intel and ARM architectures

Finally, we present results comparing runtime and energy consumption of all the main optimized configurations of our BLSTM, on all architectures. Figure 6.8 presents energy consumption measurements under different network configurations; these were collected according to the methodology described in Section 5.3.3. Runtime, in this figure, is shown on the horizontal axis using a logarithm scale, since the ARM implementations run much slower than Intel implementations. Different symbols are used to represent the different optimizations being evaluated. All optimized configurations of the network use AVX (Intel) and NEON (ARM) intrinsics for vectorization. We once again omit the

performance of the un-optimized baseline network and present just the best configurations obtained during our tests, so that they can be compared in terms of their relative performances; all of them performed significantly better than the baseline, as previously discussed.

Figure 6.8: Runtime vs energy consumption of the main optimized configurations of the BLSTM, on all architectures



It is possible to observe, in Figure 6.8, that Intel architectures provide the best runtime results. In particular, the best runtime we achieved, when considering all architectures and optimization techniques, was achieved on an Intel Xeon CPU. That implementation of the BLSTM network was capable of analyzing the entire testing set of images in just 1.3 seconds, compared to approximately 220 seconds required by the baseline implementation of the BLSTM that we optimized in this work on the same architecture. This configuration of the network is about 8 times faster than the best network we analyzed on Intel i7 CPU; this happens due to the number of available threads of each architecture. The Intel i7-4500U CPU, designed specifically for laptops, presents the best energy consumption (if executing the NN with offline parallelization using integers) across all configurations that we considered in this work; in particular, it requires 19 times less energy than the floating-point baseline network (baseline consumes 2090 joules). Compared to the floating-point implementation with parallelization, this energy reduction is of 8.5 times. Xeon E5-2670 v3 CPU reduces its energy consumption in 32 times compared to the floating-point baseline implementation (baseline consumes 8466 joules).

Despite their larger runtime, ARM architectures have good energy consumption

results, since they are designed to use very low power when executing tasks. Cortex-A9 (Zynq board) presents great energy consumption results and the most optimized version of the NN requires 5.5 times less energy than the floating-point baseline (baseline consumes 911 joules). The same comparison for Cortex-A53 (Raspberry pi 3 board) shows a reduction of 6 times of energy consumption compared to the baseline floating-point implementation (baseline consumes 1782 joules).

7 CONCLUSION

Recurrent Neural Networks, such as BLSTMs, are being increasingly used in important applications due to their superior accuracy. Current applications often require the deployment and use of these networks in devices with limited computational power. In this work, we proposed and evaluated many possible software- and hardware-based optimization techniques to speed up the execution of a BLSTM on CPUs. These optimization techniques can also be extended to other kinds of NNs, since they accelerate components that are common to all of them, such as dot product optimizations. We empirically show that by combining specific optimization methods, it is possible to speed up a baseline BLSTM implementation on different CPU architectures—by 9 times, on average, on an Intel architecture, compared to un-optimized parallel implementations, and by more than 2.3 times, on average, on ARM architectures given the same comparison. The Xeon architecture can achieve a reduction of up to 169 times comparing a complete un-optimized baseline with the best combination of optimization techniques we propose in this work. Deploying an effective combination of optimization techniques on an ARM architecture (Cortex-A53, Raspberry pi 3 model b) results in similar improvements, when compared to a baseline floating-point NN; in particular, it results in a runtime reduction of up to 9 times.

In this work we have also shown that it is possible to successfully implement, in software, optimization techniques that are typically implemented in hardware, such as numerical precision reduction and lookup tables, achieving an average runtime reduction of up to 50%. In case numerical precision reduction is used, one could expect a significant decrease in accuracy of the network, but we empirically observe that in an important application (Optical Character Recognition), the trade-off between accuracy and runtime reduction is worth it: by implementing and deploying such techniques, one incurs in a <0.32% accuracy reduction.

We have observed that performance improvements using vectorization strategies are central to achieving significant runtime improvements. However, implementing such strategies involves dealing with many trade-offs in a vast landscape of possible CPU architectures. We show that it is possible to make use of hardware-specific instructions (i.e. SSE, AVX and NEON vectorization) in order to implement critical linear algebra operations that often constitute computational bottlenecks in recurrent neural networks, and obtain better runtime even when compared to an optimized BLAS library.

Many of the optimization techniques used here perform better if used in particular combinations, instead of being deployed separately. A combination of software-based optimization techniques, associated with actively exploiting hardware capabilities, was shown to produce the best runtime improvements. Empirically, we have observed that the factor that results in largest runtime improvements is parallelization, followed by the use of SIMD instructions, numerical precision reduction, and the use of lookup tables to approximate neural network activation functions (i.e., memoization). Although memory optimizations are not mentioned above, they are one of the most important optimizations presented in this work and responsible for taking the maximum potential of all other optimizations.

By deploying the optimization techniques proposed in this work, we have shown that it is possible to perform OCR on thousands of images, with a BLSTM neural network, in just a few seconds, while undergoing negligible cost in accuracy. This is in contrast with the cost of running these same analyses but via a neural network that does not implement our proposed optimizations; in this case, it could take up to 5 minutes on Intel and 50 minutes on ARM to process the same amount of data.

Besides demonstrating that significant runtime improvements can be achieved, we have also shown that it is possible to reduce energy consumption in all evaluated architectures. The energy consumption reduction on Xeon was observed to be of up to 32 times, and up to 6 times on ARM CPUs. In this latter case, one could expect better ARMs to be more energy efficient (since they are designed as low power CPUs); however, the significant runtime gains on Intel i7-4500U architecture designed for laptops ensures that the most significant energy consumption improvements and results occurred in this architecture.

In this work we have focused on optimization methods that are software-based and/or that exploit hardware-specific capabilities. We have not, however, evaluated the possibility of using *dedicated hardware*, capable of speeding up critical linear algebra operations, such as GPUs. In this respect, we have performed preliminary experiments on a Xeon phi co-processor, which has 236 available threads for parallelization. Preliminary experiments suggest that by using GPUs (even if possible from a monetary point of view, and in devices that allow for such a hardware) may not always be advantageous. This is supported by our observation that the time required to offload a relatively small BLSTM network and training set (e.g., 200 neurons and 3401 images) to a co-processor may not be amortized by the faster processing capabilities of such a dedicated hardware. This

further supports our motivating assumption of the importance of studying optimization techniques that do not necessarily rely on this type of dedicated hardware.

We expect that the discussion presented here, about how different software or hardware optimizations can be implemented, and in which combinations, will benefit future implementations of neural networks that are required to run in general-purpose architectures (such as Intel) and in devices with important computational limitations, thereby also allowing for such machine learning methods to be more widely used in embedded systems.

7.1 Future Work

One future extension of this work relates to the idea of empirically evaluating the performance of the (un-optimized) baseline BLSTM neural network used here, but on a GPU. After performing this analysis, we could initiate efforts to adapt such NN implementation to this new target architecture, and determine how much performance improvement could be achieved when compared to the ones we have presented here—which were obtained without requiring a dedicated hardware.

Another line of future work related to trying to further improve performance by offloading the BLSTM to a hardware with more threads. As mentioned above, we have performed preliminary experiments where we offload the NN and training set to a Xeon phi co-processor, which has many much more threads than the Xeon processor. We have, however, encountered difficulties achieving high-accuracy results. An important next step could be to better understand why this is the case, and which optimizations could be deployed in order to better exploit this kind of architecture.

Finally, another important future step (after conducting more conclusive offload experiments) is the evaluation of a BLSTM NN when deployed natively on a Xeon phi co-processor. In particular, we would like to investigate what types of runtime and energy consumption improvements could be achieved if we send all data and NN parameters to the co-processor, before running the resulting BLSTM directly on the 236 threads available on this architecture.

REFERENCES

AFZAL, M. Z. et al. Document image binarization using lstm: A sequence learning approach. In: **Proceedings of the 3rd International Workshop on Historical Document Imaging and Processing**. New York, NY, USA: ACM, 2015. (HIP '15), p. 79–84. ISBN 978-1-4503-3602-4. Available from Internet: <http://doi.acm.org/10.1145/2809544.2809561>.

APPLEYARD, J.; KOCISKÝ, T.; BLUNSOM, P. Optimizing performance of recurrent neural networks on gpus. **CoRR**, abs/1604.01946, 2016. Available from Internet: <http://arxiv.org/abs/1604.01946>.

BREUEL, T. M. et al. High-performance ocr for printed english and fraktur using lstm networks. In: **Proceedings of the 2013 12th International Conference on Document Analysis and Recognition**. Washington, DC, USA: IEEE Computer Society, 2013. (ICDAR '13), p. 683–687. ISBN 978-0-7695-4999-6. Available from Internet: <http://dx.doi.org/10.1109/ICDAR.2013.140>.

BRYSON, A. E.; DENHAM, W. F.; DREYFUS, S. E. Optimal programming problems with inequality constraints. **AIAA journal**, v. 1, n. 11, p. 2544–2550, 1963.

COLLOBERT, R.; KAVUKCUOGLU, K.; FARABET, C. Implementing neural networks efficiently. In: **Neural Networks: Tricks of the Trade**. [S.l.]: Springer, 2012. p. 537–557.

EIGEN, a C++ template library for linear algebra. 2017. <http://eigen.tuxfamily.org/>. [Online; accessed in April].

GRAVES, A. **Supervised sequence labelling with recurrent neural networks.** 1-117 p. Thesis (PhD) — Technical University Munich, 2008.

GRAVES, A. et al. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In: **Proceedings of the 23rd International Conference on Machine Learning**. New York, NY, USA: ACM, 2006. (ICML '06), p. 369–376. ISBN 1-59593-383-2. Available from Internet: <http://doi.acm.org/10.1145/1143844.1143891>.

GREFF, K. et al. LSTM: A search space odyssey. **CoRR**, abs/1503.04069, 2015. Available from Internet: <http://arxiv.org/abs/1503.04069>.

HAYKIN, S. **Neural Networks - A comprehensive Foundation**. 2nd. ed. [S.l.]: Pearson Education, Inc, 1999.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural Comput.**, MIT Press, Cambridge, MA, USA, v. 9, n. 8, p. 1735–1780, nov. 1997. ISSN 0899-7667. Available from Internet: <http://dx.doi.org/10.1162/neco.1997.9.8.1735>.

INTEL Intrinsics Guide. 2011. <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>. [Online; accessed in April 2017].

Levenshtein, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. **Soviet Physics Doklady**, v. 10, p. 707, feb. 1966.

LOMONT, C. **Introduction to Intel Advanced Vector Extensions**. 2017. <https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>. [Online; accessed in July].

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MOTAMEDI, M. et al. Design space exploration of fpga-based deep convolutional neural networks. In: **2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.: s.n.], 2016. p. 575–580.

OLAH, C. **Understanding LSTM Networks**. 2015. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>. [Online; accessed in May 2017].

QIU, J. et al. Going deeper with embedded fpga platform for convolutional neural network. In: **Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2016. (FPGA '16), p. 26–35. ISBN 978-1-4503-3856-1. Available from Internet: <http://doi.acm.org/10.1145/2847263.2847265>.

RYBALKIN, V. et al. Hardware architecture of bidirectional long short-term memory neural network for optical character recognition. In: **Design, Automation Test in Europe Conference Exhibition (DATE), 2017**. [S.l.: s.n.], 2017. p. 1390–1395.

VANHOUCKE, V.; SENIOR, A.; MAO, M. Z. Improving the speed of neural networks on cpus. **Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop**, 2011.

YAN, S. **Understanding LSTM and its diagrams**. 2016. <https://medium.com/@shiyan/understanding-lstm-and-its-diagrams-37e2f46f1714>. [Online; accessed in May 2017].

ZHANG, C. et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In: **Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2015. (FPGA '15), p. 161–170. ISBN 978-1-4503-3315-3. Available from Internet: <http://doi.acm.org/10.1145/2684746.2689060>.

APPENDIX A — C++ IMPLEMENTATION OF LOOKUP TABLE

Figure A.1: Header file to use lookup tables for the neuron activation functions

```
#ifndef FUNCTIONS_HPP
#define FUNCTIONS_HPP

   #include <vector>       // std::vector< >
   #include <math.h>       // tanh, log
   #include <iostream>     // std::cout, std::cerr
   #include <string>       // std::string

   float sigmoidf(float x);

   //---------------------------------------------------------------
   // LUTs
   //---------------------------------------------------------------
   class LUTs
   {
      public:

      LUTs();
      ~LUTs();

      unsigned int ReturnNumberOfLUTs();

      void Init(std::string name, float(*ref_function)(float), float lower, float ↩
          upper, float scale);
      void Report();

      void GenerateLUTs(unsigned int size);

      float LUT_fine(float input);
      float LUT_coarse(float input);

      float func_ref(float input);

      std::vector<float> LUT;

      protected:

      unsigned int numberOfLUTs;
      float lower_limit;
      float upper_limit;
      float step;
      float recip_step;
      float scale_factor;

      std::string Name;

      float(*p_ref_function)(float);

      private:

   };

#endif
```

Figure A.2: C++ source file implementing functions to use lookup tables instead of regular neuron activation functions

```cpp
#include "functions.hpp"
#include "neuron.hpp"

   float sigmoidf(float x)
   {
      return 1.0 / (1.0 + expf(-x));
   }
   //------------------------------------------------------------
   // LUTs
   //------------------------------------------------------------
   LUTs::LUTs()
   {
      numberOfLUTs = 0;
      lower_limit = 0.0;
      upper_limit = 0.0;
      step = 0.0;
      recip_step = 0.0;
      Name = ".";
      scale_factor = 1.0;
   }
   LUTs::~LUTs()
   {
      LUT.clear();
   }
   unsigned int LUTs::ReturnNumberOfLUTs()
   {
      return numberOfLUTs;
   }
   void LUTs::Init(std::string name, float(*ref_function)(float), float lower, float upper, float scale)
   {
      Name = name;
      p_ref_function = ref_function;
      lower_limit = lower;
      upper_limit = upper;
      scale_factor = scale;
   }
   void LUTs::Report()
   {
      std::cout << "Number of LUTs in " << Name + ": "<< numberOfLUTs << std::endl;
   }
   void LUTs::GenerateLUTs(unsigned int size)
   {
      numberOfLUTs = size;
      LUT.resize(numberOfLUTs);
      // numberOfLUTs - 1 != 0
      if(numberOfLUTs - 1 < 1)
      {
         std::cerr << "ERROR: the number of LUTs is incorrect" << std::endl;
         return;
      }
      else
         step = (upper_limit - lower_limit) / (float)(numberOfLUTs - 1);
      recip_step = 1.0 / step;
      unsigned int i = 0;
      for (float x = lower_limit; x <= upper_limit + (step / 2.0); x += step, i++)
         LUT[i] = scale_factor * p_ref_function(x);//tanhf(x) or expf(x) or sigmoidf(x);
   }
   float LUTs::LUT_fine(float input)
   {
      // If we are outside of LUT range
        if (input <= lower_limit) return LUT.front();
        if (input >= upper_limit) return LUT.back();
        // Scale from [lower, upper] to [0, N]
        float xi = (input - lower_limit) / step;
        // Get left and right point of xi
        unsigned int x1 = floor(xi);
        unsigned int x2 = ceil(xi);
        // Bilinear interpolation
        float y1 = LUT[x1];
        float y2 = LUT[x2];
        return (y2 - y1) * (xi - x1) + y1;
   }
   float LUTs::LUT_coarse(float input)
   {
      // If we are outside of LUT range
        if (input <= lower_limit) return LUT.front();
        if (input >= upper_limit) return LUT.back();
        // Scale from [lower, upper] to [0, N]
        float xi = (input - lower_limit) * recip_step;
      unsigned int index = (unsigned int)xi;
        float y1 = LUT[index];
        return y1;
   }
   //------------------------------------------------------------
   // REFERENCE
   //------------------------------------------------------------
   float LUTs::func_ref(float input)
   {
      return p_ref_function(input);
   }
```