Hardware Accelerator for Recurrent Neural Network-Based Sound Synthesis

A thesis submitted in partial fulfilment of the requirements
for the award of the degree

Bachelor of Engineering (Computer)

from

University of Wollongong in Dubai

by

Manohar Vohra

Faculty of Engineering and Information Sciences

May 2019

Supervisor: Dr Stefano Fasciani

ABSTRACT

Neural Networks (NN) have created endless possibilities in the digital realms by allowing machines to be equipped with human-like cognitive skills. Computers now have the ability to perform tasks which no longer need to be explicitly programmed. Sound synthesis is a technique where analog or digital circuits are used to generate sound. NN based sound synthesis has been an ongoing area of research. Large networks are trained directly with time-sequences of audio samples rather than with extracted features from those samples. After training, these networks can independently produce new sounds with sonic characteristics similar to those used within training. Most implementations of such networks are not yet capable to generate sound in real time due to the massive computational complexity required to generate each audio sample. Providing a solution to this problem can lead to a breakthrough in sound synthesis, reimagining how sound is generated in gaming applications, cinematography, and other fields. Thus, the following research proposes the use of a high-performance platform featuring a Field Programmable Gate Array (FPGA) within a novel framework to accelerate NN kernels. A complex Recurrent Neural Network (RNN) based sound synthesis algorithm is selected as the reference application to design and evaluate the proposed hardware acceleration framework. In particular, the synthesis algorithm is profiled to identify computationally critical kernels. Kernel functionalities are then mapped to customized and parallel hardware accelerators, which are integrated, tested and evaluated against the original synthesis application. In this project, a low-cost FPGA platform is utilized and therefore real-time computation is still not be possible. In general, the framework has showcased a speed-up of the execution time of an RNN kernel by 21 times, whereas for the selected sound synthesizer, the simulation results had indicated a speedup factor roughly between nine to 16 times faster. Moreover, this has also enabled the identification of hardware bottlenecks in the platform currently used. Ultimately, the proposed framework enables custom design and easy deployment on FPGAs of accelerators for NN kernels.

ACKNOWLEDGEMENTS

I would like to take this opportunity to give my sincere thanks to Dr. Stefano Fasciani and the Faculty of Engineering and Informational Sciences at the University of Wollongong in Dubai.

I would also like to thank my family, friends, and most importantly, God, for giving support throughout this journey.

STATEMENT OF ORIGINALITY

I, Manohar Vohra, declare that this thesis, submitted as part of the requirements for the award of Bachelor of Engineering, in the Faculty of Engineering and Information Sciences, University of Wollongong in Dubai, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications or assessment at any other academic institution.

Signature:

Print Name: Manohar Vohra

Student ID Number: 5265071

Date:

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS AND SYMBOLS

ASIC	Application Specific Integrated Circuit
BPTT	Backpropagation Through Time
CLB	Configurable Logic Blocks
CNN	Convolution Neural Network
DNN	Deep Neural Network
DRNN	Deep Recurrent Neural Network
ES	Embedded System
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
HLS	High Level Synthesis
I/O	Input/Output
II	Initiation Interval
IP	Intellectual Property
LSTM	Long Short-Term Memory
MAC	Multiply-Accumulate
MLP	Multilayer Perceptron
MNIST	Modified National Institute of Standards and Technology
MOS	Mean Opinion Score
NN	Neural Network
PC	Personal Computer
RAM	Random Access Memory
RNN	Recurrent Neural Network
RTL	Register-Transfer Level
SGD	Stochastic Gradient Descent
SoC	Systems on Chip
TBPTT	Truncated Backpropagation Through Time
TTS	Text-To-Speech

LIST OF CHANGES

Section	Statement of Changes	Page Number
3.1	Added description of SampleRNN figure	24
3.1.2	Addition of new in-depth profiling.	26

1 INTRODUCTION

The popularity of Neural Networks (NN) has rapidly increased in the last two decades. They allow computers to perform tasks which do not require explicit programming. Indeed, they do so by abstracting a model from data. This, together with the large amount of data collected in the digital age, is leading to a new era of computing. Although the NN concept is not recent, the computational complexity limited NN application until recent years. Major advancements in data storage and computing architectures has enabled NNs to be used in plenty of consumer applications. Some of which include, Automotive (Tian et al., 2018), Medicine (Oktay et al., 2018), and Robotics (Levine et al., 2018).

In this work, the focus is on a novel NN application in the field of sound synthesis, a technique using analog or digital circuits to generate sound. Recently, a variety of NN architectures have been used for this purpose, providing rich, unique sounds while reducing the cumbersome process to model acoustic bodies mathematically into circuits (Wyse, 2018). Although most of these implementations in literature optimize computation performances, they are not yet capable of producing sound samples at a real-time rate.

Field Programmable Gate Array (FPGA) are chips which comprise of 'islands', known as Configurable Logic Blocks (CLBs), in a 'sea' of configurable connections (Xilinx, 2018a). The flexibility and computational power of FPGAs allow implementing any computationally expensive algorithm. Performances are typically higher than traditional Central Processing Units (CPU) as many operations can be allocated to run in parallel. These chips can be considered as an empty canvas, where endless designs can be 'painted' whichever way a developer demands it to function. In particular, hardware designed on FPGAs provide low latency, high bandwidth, and parallel execution, forming a suitable environment to implement accelerators dedicated to specific applications (Xilinx, 2018a). As seen in the literature, FPGAs have been used to speed up the computation of NNs. However, their application in the context of NN-based sound synthesis has yet to be explored. Notably, FPGAs can provide significant acceleration to the synthesis algorithm allowing real-time or near real-time execution.

1.1 Problem Statement

In context to its many applications, sound synthesis requires real-time computation, such as for digital musical instruments and interactive sonic systems. This requires computing sound samples at least at a rate matching the sampling frequency. For instance, one of the frameworks proposed by Paine et al. (2016) has a production rate of 200 samples per second, whereas the required rate for high-quality sound is usually at least 16,000 samples per second or more (48,000 samples per second). If such systems with these capabilities were to be implemented using computers of today, they would require a dedicated accelerator alongside the CPU, which alone, would not be capable of supporting such sampling rates.

To add to this problem, the existing models are benchmarked on expensive high-end PCs with fast CPUs and GPUs acting as dedicated accelerators. However, they still do not produce audio at the rate required. Moreover, in real-time musical systems, resources are limited to smaller and low-power components.

In this research, the adoption of small size FPGAs to accelerate sound synthesis is explored. With their flexible, parallel computing, and relatively low-power consumption for the performance they provide, they are ideal for this scenario. Although, when it comes to utilizing FPGAs, hardware design is a challenging aspect. Further, detailed knowledge of the algorithm is needed to identify parallelism and exploit in full FPGA resources.

1.2 Aim

The aim of this work is to accelerate a co-existing Recurrent Neural Network (RNN) based sound synthesizer while utilizing a System on a Chip (SoC) including a CPU and an FPGA. The target platform is significantly more efficient in terms of power and cost than those used in related works. Real-time performances may not yet be achieved however, a speed up factor equal or greater than what is achieved by expensive GPUs is aimed. In addition, a generic framework to accelerate other NN kernels will also be developed, while providing a simple development process to follow, essentially reducing the design-time needed. This anticipates to making FPGAs more of an option for developers seeking high performances.

1.3 Thesis contribution

The contribution of this thesis is the following:

1. A computationally efficient hardware architecture for the RNN kernel, synthesizable on FPGAs.

2. An easy to use framework for implementation of complex NNs on FPGA, based on the porting and integration of PyTorch (2018) on a PYNQ (2018) platform.

2 LITERATURE REVIEW

This chapter discusses background and existing works within the sound synthesis and NN domains. The chapter is structured as follows: Section 2.1 includes an introduction to sound synthesis; Section 2.2 introduces NNs with a focus on sound generation; FPGA-based accelerator designs are discussed in Section 2.3; finally, Section 2.4 includes a summary highlighting the limitations of the state of the art, supporting the aim of this work.

2.1 Sound Synthesis

Sound synthesis is a technique to generate audio signals with well-defined sonic characteristics using analog or digital circuits, including software running on computers. It is used in many applications such as musical instruments, text-to-speech (TTS), sound design for games and films, music, and sonic interaction.

There are many different types of sound synthesis. Traditionally, periodic waveforms such as a sine or a square wave were generated at different frequencies upon interaction with the synthesizer. This is usually coupled with a wavetable, which tabulates these periodic waves and others to playback when needed (Rise, 2014). To generate slightly more complex signals, the concept of Fourier series is applied. This suggests that any wave can be modelled as infinite summations of sines and cosines (Imperial College London, 2018), which is exactly what is done in additive synthesis (Smith III, 2018a). On the other hand, subtractive synthesis is where an oscillator signal is filtered with different cut-off frequencies until the desired output is reached.

Another technique, known as physical modelling synthesis, incorporates complex mathematical equations to model acoustic bodies for mimicking their behaviour (Hind, 2018). Earlier techniques for this included using pre-recorded audio samples whenever an event was triggered. This indeed was a simple, easy-to-compute method, although, lacked realism and quality needed. Raghuvanshi et al. (2007) investigated the real-time production of sound using a physically based model for video game applications. The proposed system works as such: many objects within the video game are modelled using a spring-mass system before-hand. This system generalizes the geometry of the objects, since different shapes lead to different types of sounds being generated. Once this operation is complete, various “modes” are calculated using sinusoids with fixed frequencies and damping ratios. The modes represent the sound to be generated by impulses on the certain mass of the object. For example, if the corner of a drum is impacted, a different sound is produced when compared to impact at the centre of the drum. During runtime, the combination of waveforms triggered are played with different ratios.

The authors then aim to optimize the above stated approach to avoid having to model many modes per object. The technique described takes advantage of human auditory perception, where it is difficult to discriminate between frequencies played nearby, especially at higher frequencies, hence there is a reduction in the number of modes that need to be played if they fall within the range humans cannot perceive. Furthermore, techniques to reduce the number of sounds being synthesized were also applied, ultimately providing higher audio Frames Per Second (FPS).

The approach used did allow an otherwise computationally expensive model to be able to produce audio at real-time. Although, the author could have also considered the phenomenon known as masking, where the audio of objects with higher amplitudes will disregard other audio samples within a nearby vicinity. Doing so will allow some audio to never be produced as it does not add to the listeners perception.

A survey done by Serra (2007) outlines the trends of sound synthesis during the year 2007 and discusses the challenges which will be come upon in the future. The author states that the go-to method for real-time sound synthesis was the digital waveguide model, which provided an efficient algorithm. The model consists of many computational physical models which include delay lines, digital filters, and non-linear elements, utilized only for certain instruments (Smith III, 2018b). Generally, the models aim to depict the acoustic waves travelling. On the other hand, both corpus-based concatenation (Schwarz, 2007) and spectral modelling methods exist where the audio is synthesized based on given descriptions using databases available. Hence, audio is simply concatenated together, producing a waveform. As future works found within the field of sound synthesis, the authors have suggested a few new techniques regarding the control, and feedback of models.

2.2 Neural Networks

A Neural Network (NN) is a computational structure mimicking the human brain. Introduced in the early 1940s by McCulloch and Pitts, NN contain layers of neurons, namely input, hidden, and output layers. These are interconnected together to form a mesh network, where each connection has a weight associated with it, as illustrated in Figure 1.

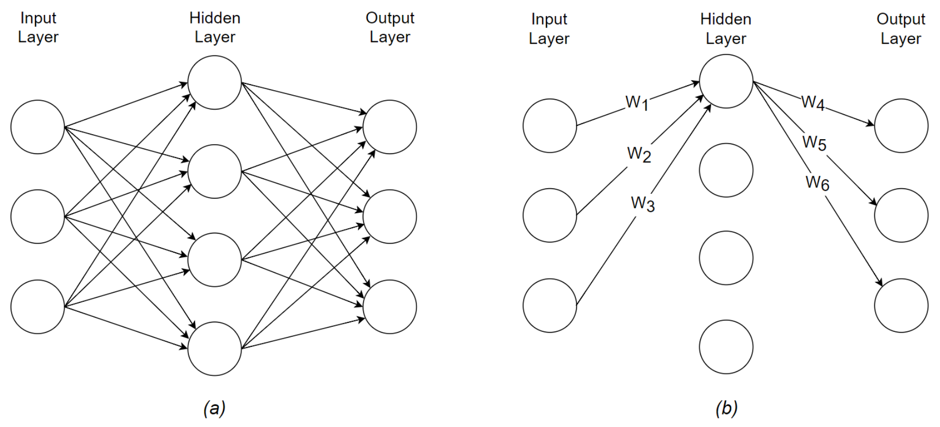


Figure 1 - (a) An example of a fully connected Neural Network. (b) A simple illustration of neuron connections to the first Hidden Layer neuron including the weights of the connections.

The weights are multiplied with the input to this connection and given to the next neuron. Neurons contain non-linear activation functions with thresholds which indicate whether or not a certain characteristic has been identified from the input. This is visible in Figure 2. During the training phase, a large set of inputs along with the desired output to the NN are presented, allowing the network to estimate the model (if any) embedded in the data (Li, 2017). The threshold (or bias) and weights are adjusted during this period, aiming to produce an output aligned with the ideal output presented.

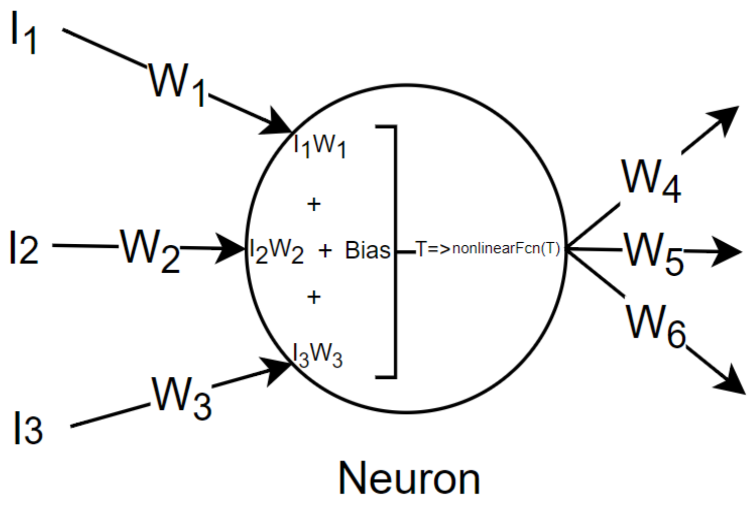


Figure 2 - Operations within a neuron. I_1, I_2, and I_3 are the inputs to the connections coming to this neuron. W_1, W_2, and W_3 are the weights of the input connections. The inputs are multiplied and summed with a bias in the neuron. This value is evaluated upon being processed by a non-linear function (usually a sigmoid or tanh).

Many NN architectures have been proposed including, but not limited to, Deep Neural Networks (DNN), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN). The difference between the networks usually relates to their structure, arrangement of the neurons, and training procedure.

DNNs are similar to regular NNs but consists of multiple hidden layers instead of simply one. They are known to provide high performance when it comes to building complex models and perform feature extraction. A drawback of such networks is that they require lots of time in training, although with the rise of powerful processing, these networks are seemingly more practical. On the other hand, CNN comprise of convolutional layers which perform filtering using the convolution operator followed by the layers of a generic NN (Stanford University, 2018) used in image applications.

2.2.1 Recurrent Neural Networks

RNNs have successfully been used in sound synthesis, and therefore extremely relevant for this work. They provide the ability to use machine learning for applications where sequential outputs are needed one at a time. It consists of short-term memory modules and many feedback loops, allowing data to be correlated. A simple overview of the RNN system can be thought of as a regular neural network with a feedback as seen in Figure 3a, however, when unrolled, the system is broken down into many subsystems shown in Figure 3b. These cascade subsystems communicate to their successor the output of that timestep. The next subsystem now has two inputs: one corresponding to the current input, and secondly, the previous output.

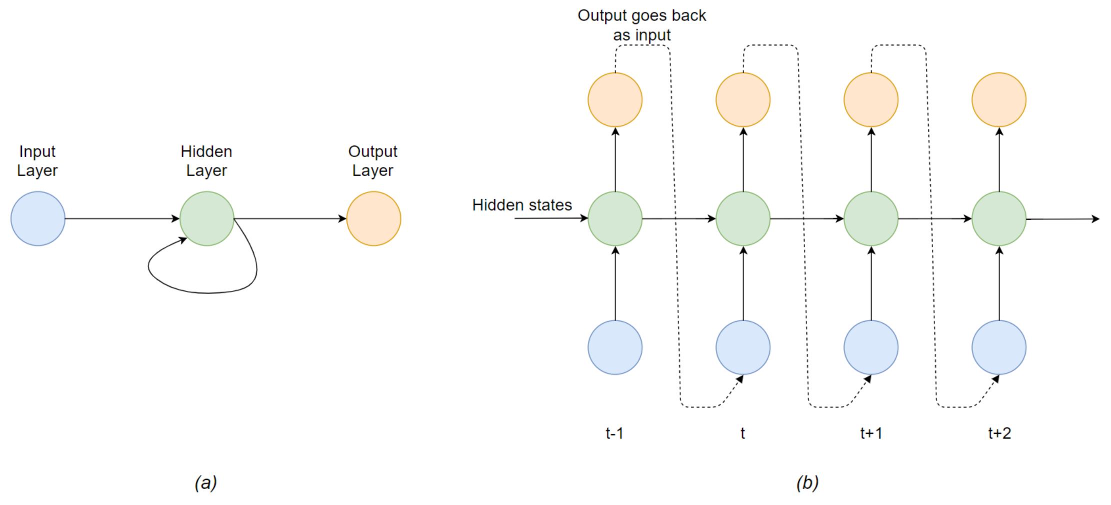


Figure 3 - (a) Basic concept of an RNN. (b) If (a) was to be unrolled in time, this will indicate how sequences are generated and how information progression occurs. Moreover, (b) clearly showcases the BPTT occurs when the weights are adjusted throughout these layers.

During the training phase of a regular NN, the weights are adjusted through verifying whether or not the output is valid. Once this is decided, the network provides a feedback, known as backward propagation, essentially changing the weights by performing partial derivatives of the error with respect to the weights. Following this, the gradient descent algorithm is used to make sure the weights are updated only to minimize the error. Within RNN, a mechanism similar to this, labelled Backward Propagation Through Time (BPTT) is actioned. Here, since the network consists of many subsystems, when the feedback is given, it can be thought of as propagating through time until the very first subsystem (Johnson, 2017).

A problem related to RNNs is the vanishing effect of the gradient, whereby the network does not learn anymore due to the gradient being small, leading to no updates being made. Usually, when the gradient is calculated, BPTT leads to the multiplication of the gradient value by many matrices, causing the value to sometimes be very small. Similarly, the opposite effect may occur, where the values are extremely large, known as exploding gradients. This results in RNNs to have degraded performances and instability. A technique to resolve this issue is with the addition of Long Short-Term Memory (LSTM) modules. These replace the traditional short-term memory modules with slightly more complicated ones, which ensure to store important information for long, providing abilities to incorporate long-term dependencies.

Setting aside the application studied by Graves (2013), a study of how complex sequences may be generated using LSTM RNNs is shown. The author particularly explained how RNN are ‘fuzzy’, as they do not simply generate outputs based on given training data but instead look more into the dependencies amongst data. Furthermore, having a large number of inputs (high dimensional) makes the network computationally expensive, but since RNNs are ‘fuzzy’, they do not encounter this. The LSTM cells, shown in Figure 4, allow the network to handle what is being stored in the cell with the help of another set of activation functions. It also is able to identify whether the stored information should affect the current time sample through the output gate, whereas the forget gate decides if at all this information should be kept any longer.

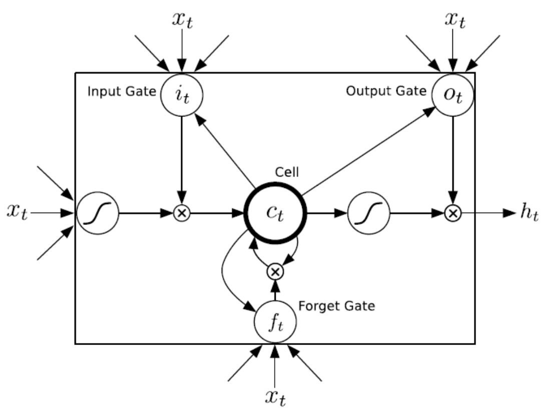


Figure 4 - The LSTM cell (Graves (2013))

The author was able to generate complex sequences with credible results. With the feasibility of using such resource-expensive modules (LSTM cells) in question, they do pose as an essential component within RNNs.

Furthermore, Chang et al. (2017) studied an extreme case of these problems through the generation of long sequences. This involves having to include extremely long-term dependencies, while maintaining short- and mid-term memory. The proposed architecture, labelled DilatedRNN, is a composite of skipped connections allowing fewer parameters, and a hierarchical design looking to store different dependencies through time. Further analysis on the dilated connections indicates that, when compared to regular RNNs, connections moving from one timestep to another are skipped by some proportion. Doing so will allow a significant improvement in speed through parallel execution, which lacks in regular RNNs. This is done by no longer having many interconnections, which usually implies that for the model to proceed in processing to the next timestep, the previous ones must be complete, else no input will appear to the current timestep. Instead, now that there are fewer dependencies, executing the various connected cells simultaneously improves performance by s times, where s is the number of skipped connections. In order to maintain long-term dependencies and avoid vanishing/exploding gradients, the architecture deploys the use of multiple hidden layers which have the parameter s increasing exponentially. Hence, it can be said that each layer/hierarchy takes into account different ranges of dependencies, where lower values of s incorporate short-term dependencies and vice-versa. Results obtained through this methodology were competitive, however, the model achieves desired results only in some applications.

Further, it is detailed that speed performances have been accelerated although it is unclear how long does it take for execution when compared to other models running on the same system.

2.2.2 Co-existing NN-based Sound Synthesis

Oord et al. (2016) proposed a neural autoregressive generative model, WaveNet, in a study to evaluate its performance for raw audio signal generation. The study specifically aimed to tackle the requirements posed by audio synthesis being a demanding task in terms of temporal resolution. Simply put, sound synthesis requires at least around 16,000 samples per second to be generated to produce decent quality output, whilst ensuring that the samples are correlated at different granularities.

The model utilizes dilated causal convolutions, a method which can increase the receptive field (amount of connections from the input to the next layer. Also known as the filter size) without having to increase the computational cost significantly. It is important to have a larger receptive field so that more features can be extracted and considered. In addition, since each audio sample is stored as a 16-bit integer, and since the next audio sample must be generated by the previous ones through a probabilistic algorithm, the authors have used mu-law companding for compression, which makes the possibility of an audio sample from being 2^{16} different values to just 256. This reduced the number of outcomes of the probability, making it easier to work with. Furthermore, additional parameters which alter the characteristics of the audio generated can also be provided to the model.

Thorough testing had been conducted, where the Mean Opinion Score (MOS) was calculated based on listeners giving scores for sounds generated by the proposed model and other models. Results clearly indicated that the model proposed was preferred and thus proved that the autoregressive model can be used to implement such applications. On the other hand, another outcome of this work was that tests done on music generation indicated that the receptive field was vital, as the audio will not be acoustic otherwise. For instance, the current sample and previous many samples must have correlation when it comes to music, else random tones will be synthesized. All in all, Oord et al. (2016) addressed many challenges faced during the design of the model. They provided reasonings for their considerations for most aspects of the design, comparing alternatives as well. Moreover, the authors do not analyse the rate at which sound is synthesized.

Paine et al. (2016) proposed an algorithmic enhancement of the previous framework which drastically reduces the computational complexity of the network. In general, the authors took a deeper look at the redundant calculations being conducted in the original approach and replaced this with memory caches in the form of a queue. This then allows the network to avoid recalculating certain parameters during runtime, reducing the computational complexity from $O(2^L)$ to $O(L)$, where L denotes the number of layers. The way the queues operate is, due to the original model utilizing dilated convolutions, outputs of each layer will be based on the ones many timesteps behind. Therefore, the queues will have varying length based on the dilation at each layer. Since the original model had an exponentially increasing dilation, the length of the queues will also exponentially increase as 2^l, where l is the hidden layer number.

In addition, the behaviour of the queue is first-in-first-out. This facilitates the fact the data stored initially will be of timesteps further away and is needed to be taken into account at first. Moreover, remodelling the system with the above stated changes allows a new interpretation of the network. Instead of being a CNN, it can now be thought of a multi-layer RNN, where the inputs to a neuron is the expected input plus a parameter from the queue (popped out of the queue). In result, it provides the output which is passed onto the next layer, and a new parameter pushed onto the queue.

To verify if the new enhancements actually provide better performances, the network is compared with the original having the same trained parameters. It was clear that the new additions to the framework proves to be much more real-time oriented through its low variations when the number of layers is increased. Hence, it can be said the new model is much more predictable. On the other hand, the work presented by Paine et al. (2016) is among those measuring the amount of time taken to generate a single sample of audio. However, no details have been provided, other than the use of a GPU, on the specifications of the machine used for this purpose. Further analysis, illustrated in Figure 5, shows the frequency at which audio is produced indicating a mere 200Hz (one sample = 0.005secs with a single layer), whereas a larger model (15 layers) would produce at 100Hz and 6.25Hz for the enhanced and original approaches, respectively. These results clearly indicate significant room for improvement of the current state.

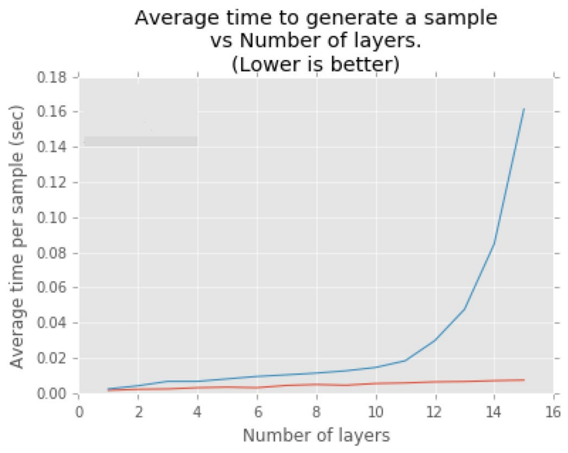


Figure 5 - Results from Paine et al. (2016). Here the Faster implementation is the red curve, whereas the blue curve represents the original WaveNet.

Another model proposed by Mehri et al. (2016), SampleRNN, investigates audio generation using RNNs. This differs from the above model such that it does not utilize Deep Neural Networks (DNNs). Further, instead of dilated causal convolutions, a technique known as Truncated Backward Propagation Through Time (TBPTT) has been implemented. The problem this paper resolves is that audio signals have plenty of samples for a single word being spoken. This implies that any kind of audio generation needs to consider the fact that to produce an audio signal which is intelligible, it must come in with many samples put together. Previous methods involved compressing the signal, followed by identification and modelling of audio dependencies. However, when the signal was decompressed, the quality of the waveform was degraded.

Therefore, Mehri et al. (2016) propose the use of RNNs to model the dependencies (feature-extraction, therefore reducing the samples that need to be generated). Although, RNNs themselves are not the best when it comes to producing a massive number of samples required in this application, the model proposed has a hierarchical implementation. Doing so compliments yet another requirement of audio generation, where dedicated hierarchies consider only audio samples with correlation with nearby samples and other hierarchies for samples far away. Similar to the work detailed in the previous subsection by Chang et al. (2017) however in another context.

Hierarchies, called tiers, each look at the input samples differently. For instance, the highest tier, where the initial input is provided, considers dependencies at frame-level. Whereas the for the lowest tier, analysis of data is considered at sample-level. Hence, each tier has its own frame size to identify dependencies from. Once the data is analysed, it is then provided to the tier below it, allowing dependencies from a large, down to a low range to contribute before producing an output.

Note that each tier will have two sets of inputs (except for the topmost, which has one), where one represents the input of this tier at a certain timestep, while the second being the modelled dependencies coming from above the tier. Flowing down the tiers, data is up-sampled to match timing resolutions, ensuring computation is mathematically appropriate. The computation done is between the current inputs multiplied by the weights, added to the data from the tier above. Followed by the up-sampling mechanism (perforated: concatenate zeros and perform linear convolution).

Since the lowest tier does not require to model long-range dependencies, the authors have utilized a memoryless Multilayer Perceptron (MLP) network. This is the generic type of NN which has been described earlier and is chosen since it does not demand as much resources as RNNs. For the output of the network, linear quantization is performed to discretize the values, followed by a softmax layer. The functionality of this layer is to map a vector into a probability distribution, where each element of the vector is given a probability. To further enhance the results here, mu-law companding, which is a non-linear quantisation technique could have been investigated, reducing the number of probabilities to consider as applied by Oord et al. (2016).

When Oord et al. (2016) designed WaveNet, they did not consider RNNs due to their prolonged training times, however, the authors here tackle this through TBPTT. Similar to BPTT but with long sequences being shortened, so the updates do not have to go as far as the beginning of the network, which they technically would have to otherwise. The authors here have presented a framework which provides competitive results, where human listeners mostly preferred the output of this model than any other compared with. The author also provided an analysis of which method to use among Gated Recurrent Units (GRUs) (explained below) and LSTM, suggesting the former performed slightly better. On the other hand, the author has not considered providing timing performances, failing to state whether audio production is fast enough.

The inputs provided for training a NN-based sound synthesizer are critical information on the desired sound (i.e. the pitch, amplitude, instrument type) and an audio sample. Wyse (2018) studied the effect of data-driven sound synthesis models with real-valued parameters. The model requires an initial input audio sample concatenated with parameters, ultimately forming a vector. Once provided, the model continues to generate audio samples (previous output is current input). Though, at each timestep, the parameters are continuously provided, giving an opportunity to change sound characteristics during runtime.

In addition, the net also includes GRUs. These are like LSTM cells, with a similar aim to resolve gradient problems with RNNs stated earlier. GRUs can selectively store dependencies from different time instances although it does not contain memory cells. They are far less complicated in design and outperform LSTM cells in certain applications (Chung et al., 2014). With regards to the design proposed by Wyse, there exist a four-layer RNN with GRUs, all consisting of feedback connections to itself. Both input and outputs are processed through linear mappings, where the vectors are decoded and encoded, respectively. Since the training parameters are implicit (many parameters included rather than primarily one, hence model considers multiple input variations) and not conducted for all the possibilities of each parameter, the challenge would be how the model will react to a certain parameter changing.

The author only provides an overlook on the pitch parameter which, when trained on two extreme pitches only and then provided inputs of linearly increasing pitch between these two points, provides results which successfully interpolate between the two trained points. But it is worthy to mention that performance is much better when near to trained data. Furthermore, another challenge within such an architecture is to do with the responsiveness of the network to changes in the parameters. A problem seen in previous models by the author were discrepancies in desired output with small errors accumulating due to large variations in the input when compared to the trained inputs. But the method chosen here does not allow such effects to occur. Once again, a model is trained with extreme pitches. When parameters are changed during runtime, and the new inputs which are never seen by the network are passed, the system was able to react and provide meaningful alterations.

The author does not investigate other parameters being changed but does mention that the pitch is something rather perceived than seen clearly (never perfect periodicity). This makes the pitch an attribute harder to correlate. On the other hand, as future works, the author claims that the system is not yet real-time however, fails to mention any quantitative statistics.

In another attempt to improve the speed performances of WaveNet, Kalchbrenner et al. (2018) have synthesised WaveRNN. The authors identified an equation accounting for the time taken to generate audio samples and thus tackle each variable to achieve real-time capabilities:

$$\boldsymbol{T}(\boldsymbol{u}) = |\boldsymbol{u}| \sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{N}} (\boldsymbol{c}(\boldsymbol{op_i}) + \boldsymbol{d}(\boldsymbol{op_i})) \tag{1}$$

T(u) is the time taken to produce ***u*** samples per second. Each sample consists of processes occurring at ***N*** layers, where the time taken by operations within each layer can be broken down into computational (***c(op$_i$)***) and overheads (***d(op$_i$)***). Computational time is the actual time taken for executing operations (i.e. an addition). Whereas, the overhead would be the time taken in invoking the operations (i.e. calling the function to perform addition). One of the major changes within the proposed model here is that it utilizes a single-layer RNN instead of the original CNN within WaveNet. Doing so enables reduction in the value of N by nearly 12 times, without degrading sound quality. On the other hand, computational costs can be reduced when the parameters in the network are reduced. This is fulfilled by a technique known as weight pruning (Narang et al., 2017), where analysis of the network weights for recurrent and linear layers is conducted whereby elimination of the weights contributing the least to the outputs is actioned, resulting in a smaller, sparse network. Lastly, the term ***u*** representing the samples produced must also be minimized as it is multiplicative term in the time equation. For this, the author proposes sampling in batches, where the dependencies will occur among batched samples which are much fewer than otherwise.

With the above changes, the network uniquely becomes much more suitable for an embedded system than ever before as there is a reduction in memory and computational costs. The author also provides information on running such a model on mobile CPUs proving its effectiveness, achieving 19,800 samples/sec (minimum). Additionally, overhead costs are minimized through enabling GPU usage within the TensorFlow library (TensorFlow, 2018). Hence, with the upcoming of GPUs on mobile devices, the proposed model can lead to many new applications of similar kind. The work exclusively situates itself as one of the only real-time models, although it would be noteworthy to test the system architecture to produce raw audio rather than simply Text-to-Speech (TTS).

2.3 Hardware Accelerators

Hardware accelerators are when special hardware devices, such as GPUs and FPGAs, are used to perform tasks outside of the CPU to provide faster execution. Moreover, hardware accelerators also prove to be effective when it comes to power consumption and large data bandwidth (Intel®, 2018).

2.3.1 Hardware Accelerators for Neural Networks

Since NNs are computationally expensive algorithms, especially with larger models, the use of hardware was expected to accelerate them. Works presented by Oh and Jung (2004) demonstrates an implementation of NN-based text detection system on GPUs. The authors look at taking the most repeated operation of a NN, which in that case was the multiplication between an input and a weight with an addition of the bias, placing them in three separate matrices for each layer. These variables are first transferred to the GPU. Next, the equation is computed in one-shot, taking advantage of the repetitive nature and parallel computation possible. With a simple test run, the GPU provided a 20-fold increase in performance. Although the results are not shown with extensive details, nor have many test cases been analysed. Overall, a simple concept of how GPUs can be included within NNs has been showcased.

2.3.2 Field Programmable Gate Array-based Accelerators

An alternative to using GPUs are FPGAs. These are reconfigurable hardware that, off the box, do not provide any functionality. They are required to be programmed (design hardware and download bitstream) in order to implement a specific computational architecture. For development purposes, FPGAs are well-known for providing a cheaper, quicker (production, since they only need to be designed and uploaded to the chip), but slower and more power-hungry alternative to Application Specific Integrated Circuits (ASICs).

Chang et al. (2015) propose a RNN-based implementation on an FPGA. RNNs are difficult to accelerate as they are naturally sequential, however, due to their popularity, there exists a need to investigate optimizations for such networks. The model created includes the usage of a two LSTM layers and 128 hidden units. As for the architecture design here, the LSTM cells do not look at the stored data to decide whether it must be forgotten or not. Instead, there are many functions which control this. In hardware, this module consists of a three gate and one element-wise submodules. The gate submodule consists of Multiply-Accumulate (MAC) blocks which perform the multiplication and addition of the weights being multiplied with the input. This is performed for the current timestep and previous output simultaneously. Following this, the outputs are added and passed on to the non-linear function blocks, which compare if the result activates them or not. This information is then passed on to the next layer and the process is repeated. Data coming in is taken from the Direct Memory Access (DMA) block which needs to be synchronized using buffers. Another aspect which must be considered here are the datatypes. For the inputs of the LSTM cell, the data is brought from 32-bit to 16-bit, but after the multiplications, the datatype is 32-bit once again. The data format of the 16-bit numbers are fixed point, with 8 bits for both the fractional and integer parts.

Once the data is calculated from the gate submodule, it is passed to the element-wise submodule, where the final output is calculated along with a control signal. The control signal decides whether the output must be stored or not. The module is synchronised into sequential stages, which utilize parallelism within. Among the stages, signals are stored as internal vectors for the next stages. The network, due to the limitations on datatype, is not as accurate as other implementations. However, other FPGA models showcased far worse performances due to using floating-point. Results shown regarding the execution time are vague as they only represent operations per second. A gap in this architecture would be that sequential stages were still used. If it is possible to eradicate this, far better performances can be achieved. On the other hand, the proposed solution shifted the complete network onto the FPGA, instead, a hybrid design involving the use of a CPU plus FPGA can improve results (CPU does perform certain tasks much better).

Huynh (2017) states that for the implementation of DNNs, general purpose hardware is not suitable since they are slow, expensive and have a high power consumption. Instead, the use of reconfigurable FPGAs is applied. Designing dedicated hardware for this include considerations of datatypes for the inputs and weights, and how these will be stored in memory. These factors ultimately decide how fast and accurate the network will be. The framework proposed provides the user the opportunity customize the structure of the network to their own needs through changes made in the VHDL program. It uses 16-bit half-precision floating point for data/weights within the layers of the network. It works such that there are control signals which indicate which layer is being computed currently, and so data is pulled out of memory for that layer. The network is formatted such that there are many neuron blocks in the network. Within these neuron blocks, the first step conducted is the input multiplied by the weight of the connection to the neuron. After this, the activation function is evaluated. These steps within the neuron block are conducted sequentially. Since a layer has many neurons, the many blocks, in parallel, perform this execution.

Testing is done using the Modified National Institute of Standards and Technology (MNIST) hand-written dataset. The system was able to gain high resource utilization, while maintaining accuracies. It is vital to mention accuracies here as the datatype used will directly affect this. There were other models which had better performances, but the design here proved to be easier to use in terms of training. On the other hand, the author only investigates fully-connected DNNs, having only a single physical computing layer. In addition, the author mentioned about being power conservative however no results have shown how this compares to other implementations. Moreover, a possible pre-fetch of data can also be considered while computation is occurring to avoid waiting for data to be pulled. Also, as additional results, performance in terms of speed would provide further analysis on how the model developed competes. Lastly, the design methodology looks at simply implementing the complete network on the FPGA but there might be some operations that may be better to have implemented on a CPU, thus creating a hybrid model, as mentioned earlier. This of course requires further in-depth as to what actually are the bottle-necks within DNNs.

In contrast to the above work, Guan et al. (2017) study the runtime acceleration of LSTM-RNNs on an FPGA. The authors do perform profiling of a selected application to analyse segments of the network which are bottle-necks currently. Results indicate that the program spends most time inside the LSTM cells which have a complex algorithm as mentioned earlier. Another aspect of the network which requires enhancing are the activation function, most of which either involve division or complex mathematical operations (i.e. tanh). The enhancements performed for the above two problems is as follows. The LSTM cells are broken down into smaller modules within the cell, allowing the individual components to execute in parallel. Data coming in and out are buffered and distributed using crossbar technique. With regards to the activation functions, these are replaced by simpler operations such as addition and shifting which mimic the complex non-linear functions. The approximation of the functions does lead to inaccuracies, although when calculated, the authors found subtle differences.

Another key issue when designing hardware is the communication between memory modules and the FPGA. This usually adds to the overhead cost of the network. To tackle this, the authors include revamping the matrices in memory so that the accesses are fewer and in a regular pattern. Moreover, a data dispatcher is designed to improve bandwidth. For this purpose, the AXI4Lite bus (Xilinx, 2011) is used, which is a communication interface. The model was then tested, providing up to 20 times faster execution time. The resource utilization was moderate. Comparison with another model were inconsistent as the size of the networks differed greatly. As a deeper analysis, the author identified the LSTM cells as a crucial factor in execution of the network, however, an alternative mentioned earlier was the use of GRUs, which are essentially far less complex. It would be worth testing the performance with GRUs and seeing the speed-up factor, if any. In addition, AXI4-Steam interface was not considered by the authors. These provide means through which larger amount of data to be transmitted together (Xilinx, 2011). Lastly, the model uses 32-bit floating point datatype, which is expensive to use but would definitely provide high accuracy, which were not recorded.

Hao and Quigley (2017) have also designed an accelerator on an FPGA but for Deep RNNs (DRNNs), aiming to prove how the FPGA-based technology can be brought into Embedded Systems (ES). The reason why this research is significant is that most implementations of NNs have often been implemented using high-end CPUs and GPUs. The case with ESs is that they do not have this luxury, nor do they have soft power or memory constraints, leading to fewer DNN implementations. The proposed solution involves building a DRNN using Theano, a library which the author claims to be most suitable for a 32-bit Operating System (OS). It also requires less memory than alternatives which is key for the purpose. The board used for the implementation is the Xilinx PYNQ-Z1. This uniquely provides an dual-core ARM Cortex A9 microcontroller simultaneously with an FPGA (Digilent Inc, 2018). The combination of being able to develop Python applications and include hardware acceleration together makes this platform extremely powerful.

With respect to the NN design, the model houses three hidden LSTM layers. The computation within these layers to drive the logic is programmed through Theano. Interconnections among these layers are as discussed before although at the output layer, the use of a softmax function is actioned. To train the network, the error calculation is done through the cross-entropy function and applied using the Stochastic Gradient Descent (SGD) method. This is exactly what was explained earlier regarding gradient descent, where partial derivatives of the error are taken with respect to the weights and adjusted. As for the accelerator, the hardware designed performs matrix multiplications and addition, as seen in designs above as well. To further increase the performance capabilities, the design incorporates five processing elements as the matrices are large. Unlike the previous work, the authors here have included the AXI Stream for batch data transmission from the memory to the FPGA. Hence, having multiple processing elements and providing large data through this interface provide a design which goes hand-in-hand.

Overall the accelerator can perform 2500 additions and multiplications each, while having a latency of only 250ns. Moreover, the authors do not highlight how the datatype, which was fixed point, selected (only mentioned in results) affected the accuracy of the overall model. Also, no information had been provided on the structure of the datatype such as how many bits were allocated for the integer and fractional parts. Furthermore, the hardware design could have included many other segments of the NN, perhaps profiling and further investigating the LSTM modules. Note that the work mentioned earlier did state that most time was spent within the LSTM cells, which contradicts the work here. Since the authors aimed at bringing such architectures to ESs, a timing analysis other than operations per second would definitely analyse whether this was achieved. In addition, lack of depth was shown in to how the Overlays (library to communicate with hardware design) were embedded into the design. Lastly, the authors fail to recognise other NN libraries such as PyTorch, which indeed is powerful especially when using Python.

2.4 Critical Review

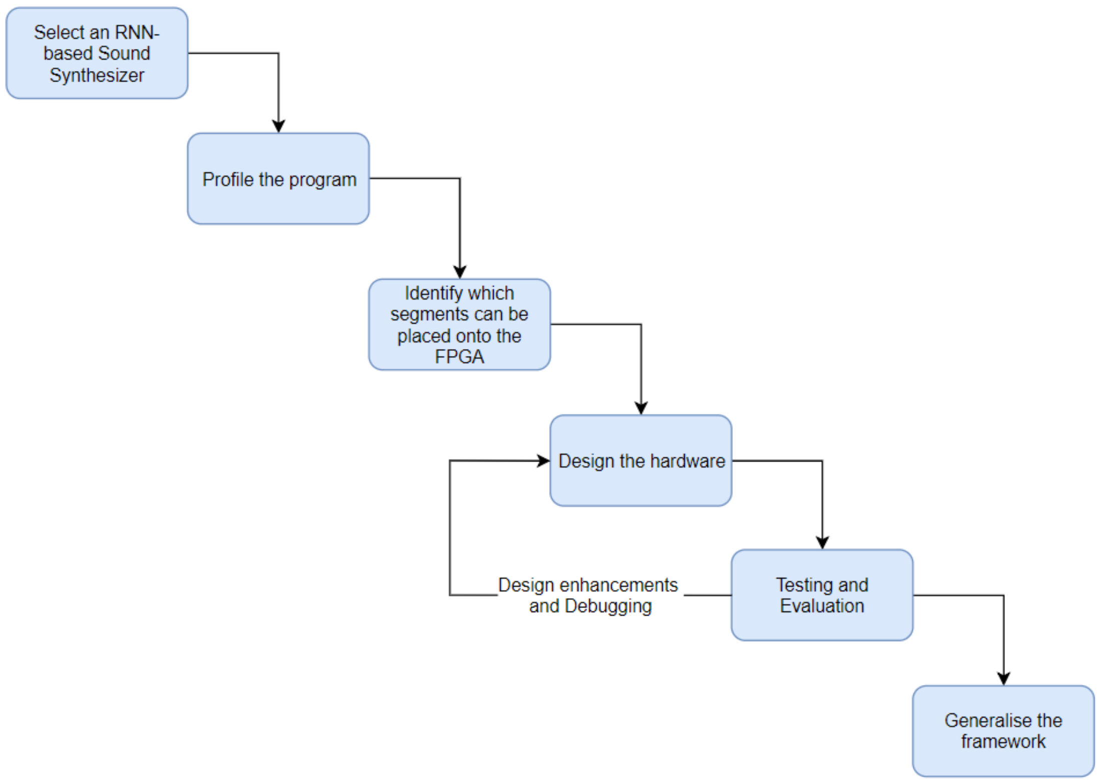
The literature reviewed in this chapter has provided an extensive breakdown on the current state-of-the-art models. As future works found within the field of sound synthesis, a few new techniques regarding the control, and feedback of models can enhance performances. Moreover, sound synthesizers must look at the auditory perception of humans to eradicate generation of certain waveforms within algorithms, essentially reducing the computations required. In terms of NN implementations of sound synthesizers, a common gap seen is that authors do not express how much time, approximately, it takes to generate a second of audio, similar to what was done by Paine et al. (2016). Results shown by this author clearly indicated that even the model deemed faster than one of the state-of-the-art can merely produce 200 samples per second which is extremely slow. Although time performances are platform dependent (i.e. faster CPU provides faster execution), authors can execute the work they compare on the very same platform and mention the specifications of it, informing the reader or other researchers how the results were obtained. This will provide a fair and informative conclusion. Moreover, the FPGA designs studied included many sequential stages (i.e. Huynh (2017)), which indeed can be remodelled or executed on a CPU as a hybrid architecture. Usually, when hardware is designed, synchronisation and communication become key factors which must be taken into account for the success of the design. Indeed, these factors add to the complexity of using FPGAs and other hardware as some techniques are not conspicuous, requiring expertise.

3 METHODOLOGY & DESIGN

This chapter details on the methodology to systematically approach the problem discussed in the previous sections. First, we provide a high-level description. This is followed by an in-depth look into the system design highlighting its robustness.

3.1 Methodology

The proposed design aims at enhancing the speed at which audio is generated by an RNN-based sound synthesizer. A breakdown of the approach is shown in Figure 6.



Select an RNN-based Sound Synthesizer
Profile the program
Identify which segments can be placed onto the FPGA
Design the hardware
Testing and Evaluation
Design enhancements and Debugging
Generalise the framework

Figure 6 - Flowchart of the methodology.

To begin with, the model chosen is a PyTorch implementation of SampleRNN coded by Kozakowski et al. (2017). The system was originally designed by Mehri et al. (2016) as examined earlier which consisted of a hierarchical design, as seen in Figure 7.

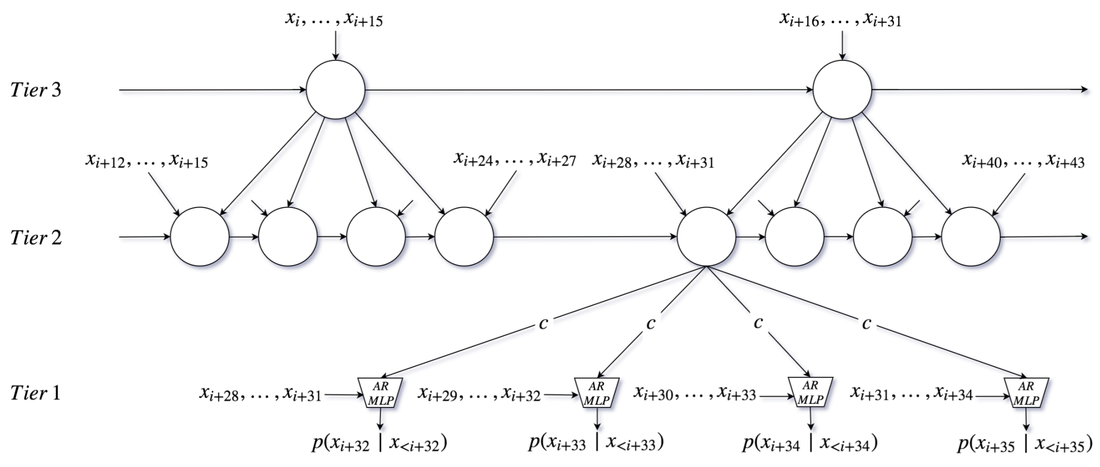

$x_i, \ldots, x_{i+15}$
$x_{i+16}, \ldots, x_{i+31}$
Tier 3
$x_{i+12}, \ldots, x_{i+15}$
$x_{i+24}, \ldots, x_{i+27}$
$x_{i+28}, \ldots, x_{i+31}$
$x_{i+40}, \ldots, x_{i+43}$
Tier 2
c
c
c
c
Tier 1
$x_{i+28}, \ldots, x_{i+31}$
AR MLP
$x_{i+29}, \ldots, x_{i+32}$
AR MLP
$x_{i+30}, \ldots, x_{i+33}$
AR MLP
$x_{i+31}, \ldots, x_{i+34}$
AR MLP
$p(x_{i+32} \mid x_{<i+32})$
$p(x_{i+33} \mid x_{<i+33})$
$p(x_{i+34} \mid x_{<i+34})$
$p(x_{i+35} \mid x_{<i+35})$

Figure 7 - SampleRNN unrolled design by Mehri et al. (2016)

As the network unfolds from Tier 3 down to Tier 1, the memory granularity increases or, in other words, the scope at which sound is correlated changes from a broader range right down to samples near each other. The broader range tiers consist of GRUs whereas the sample-level tiers are convolutional layers.

In order to accelerate any model successfully, it is essential to identify the bottlenecks of the system. To perform such analysis, the program implementing the RNN-based synthesis must be profiled. Profiling is a technique monitoring code execution, resulting with details on how much time is spent in each function and how many times the function is called. For the current scenario, profiling must be conducted on the program which produces the audio only (i.e. network inference). Once we train the network, the segment generating audio can be isolated and profiled, where the functions requiring the most amount of time to execute will be further analysed. It is important to understand the functionality of the time-consuming commands as its feasibility on the FPGA will be in question. For instance, if a command is found to be accountable for 90% of the execution time, however, if it cannot be recreated on the FPGA to mimic its behaviour, significant acceleration will not be achieved. Therefore, the commands must be taken to the lowest level possible (function calling another function and so on) to be able to judge this.

Once the analysis is complete and it is known what operations should be designed on the FPGA, it must be implemented using one of the four methods as follows. First of which involves using Vivado HLS by Xilinx to create an Intellectual

Property (IP) using C Synthesis. On the other hand, another technique considers implementing a visual design using System Generator (another tool by Xilinx) which will also be exported as an IP. Moreover, a combination of these two methods can also be incorporated. Both these tools are available in the Design Suite by Xilinx (Vivado, 2018). Alternatively, the design can also be programmed in VHDL. More details will be given on these in upcoming subsections.

Regardless of the technique applied, the designed components will be continuously verified. Testing will be done to ensure that the model designed in hardware provides the correct output for a set input, whereas evaluation will also then be conducted to measure the speed-up factor.

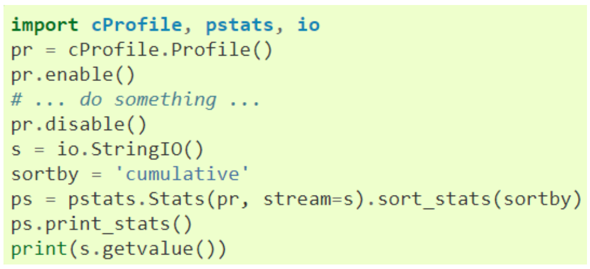
To be able to generalise the framework which will accelerate any RNN-based application, all design considerations must be taken into account. This will allow developers using this framework to understand what steps will make their designs perform better. Furthermore, FPGA design is challenging and so the proposed framework integrates an existing deep learning programming environment with FPGA deployment tools. This simplifies designing and deploying accelerators for NNs, allowing developers to implement and evaluate application-specific accelerators.

3.1.1 Model Selection

The model selection was done after attempting to execute source codes of various models discussed in the literature and testing them on a PC. SampleRNN was one of the models which provided great flexibility in the way the program can be used. For example, multiple flags can be provided to alter the size of the network, whether to use CPU or GPU and so on. This implies that the model can be tweaked with ease without having to modify the actual code. Although, when required, the program is packed with many classes which can be used to achieve more personalised requirements. Furthermore, preliminary results indicate that the model was able to produce audio at 49.3Hz and 441.5Hz on recommended settings, using CPU and GPU respectively. The results are clearly evident that there is a significant amount of room for improvement.

3.1.2 Python Profiler

The purpose of a profiler is to provide statistical information for a piece of code upon execution. This information may include, the amount of time spent at a command, the amount of times a function was called and more. ‘cProfile’ and ‘pprofile’ are profilers recommended by the Python documentation (2018). The former provides an easy-to-use, lightweight mechanism (less overheads, much better for embedded platforms) which will be utilized within the analysis here. An example is shown in Figure 8. Between enable and disable, the function calls for generating audio will be placed.



```
import cProfile, pstats, io
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

Figure 8 - cProfile example taken from Python Documentation (2018).

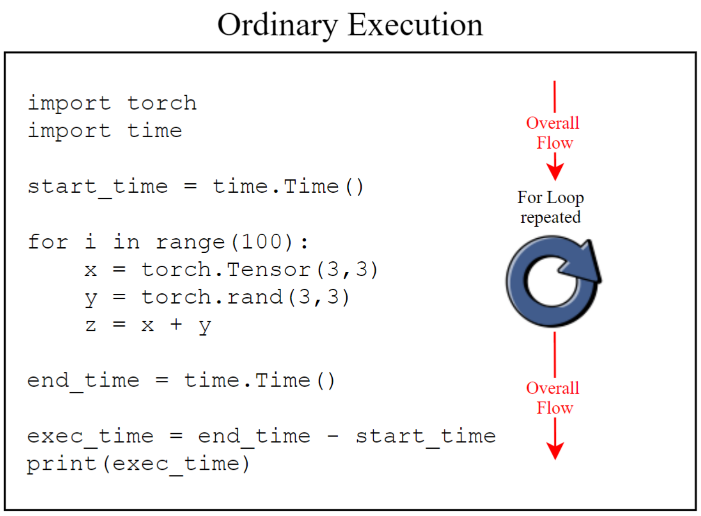
In Figure 8, alongside ‘cProfile’, another library known as ‘pstats’ is also added. This allows developers to manipulate and showcase data in different methods. For instance, it is currently set to sort data by the cumulative time spent (usually the case when a function is called more than once). In addition, the ‘io’ library allows the pstats-manipulated data to be stored in a string, which is later displayed. Note that the profiler itself does provide raw data (ps.print_stats()).

Once this top-level analysis is conducted, the results help narrow down the search for bottlenecks from the complete program down to functions used within. Although, most functions will perform a variety of steps before returning, simply knowing which function occupies most of the CPU time within a program is not sufficient to identify the actual bottleneck just yet, but a step closer is accomplished. Hence, once the top-level profiling has been completed, an in-depth look into those time-consuming functions are necessary at line-level. This can be done using a tool developed by Kern (2019), wherein handles are placed just before the function bodies of whichever function is being aimed to be further studied. The tool is then invoked on the python program using command-line, where execution and monitoring begins, finally providing results which dictate how much time was spent, in seconds and as a percentage, at each line of a function throughout the program execution. Doing so will allow to keep tasks which are indeed handled by the CPU far better to not be ported onto hardware, which will save development time and resources available on the FPGA chip.

3.1.3 Accelerator Design

The proposed framework will be based on the Xilinx PYNQ-Z1 board (Digilent Inc, 2018). The Systems on Chip (SoC) uniquely provides a single platform solution containing both a microcontroller, and an FPGA. Integration between the programmable (RISC microcontroller) and reconfigurable (FPGA) components is done through a concept of a software library labelled as Overlays. In conjunction with a powerful Python developing environment and a Debian-like Operating System, the board places itself perfectly for the purpose. The board has been used to accelerate Deep Recurrent Neural Networks by Hao and Quigley (2017). What the authors of this paper have failed to take into account is one of the most competitive NN libraries, PyTorch (2018a). This research will also be the first to execute PyTorch-written code on the PYNQ-Z1, which is expected to open a gateway of new applications and their enhancements.

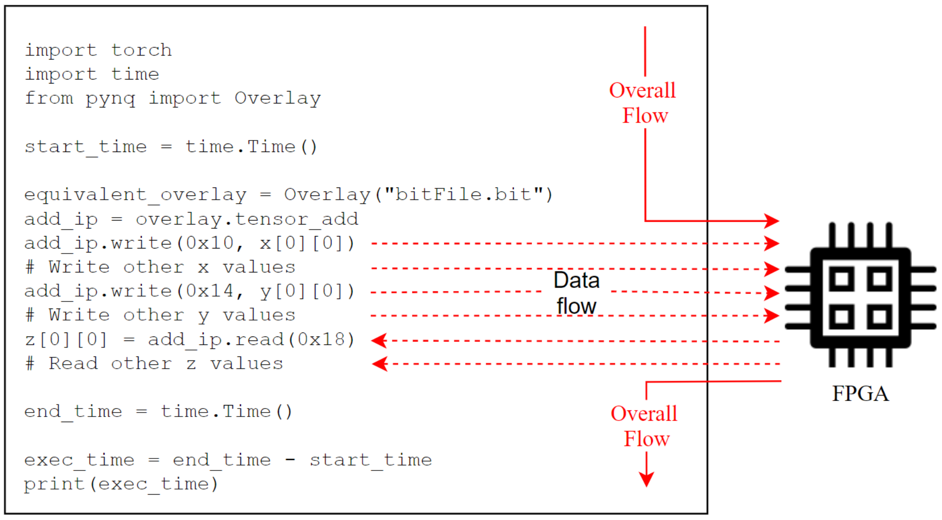
Figure 9 and Figure 10 compare regular execution with the one proposed here. In particular, Figure 10 illustrates the hybrid runtime execution behaviour of the framework. Note that a hybrid design is chosen as some operations perform much better or are convenient to do on a CPU than an FPGA.



```
import torch
import time

start_time = time.Time()

for i in range(100):
    x = torch.Tensor(3,3)
    y = torch.rand(3,3)
    z = x + y

end_time = time.Time()

exec_time = end_time - start_time
print(exec_time)
```

Figure 9 - Ordinary Execution with any CPU/GPU implementation.

Proposed Execution



```
import torch
import time
from pynq import Overlay

start_time = time.Time()

equivalent_overlay = Overlay("bitFile.bit")
add_ip = overlay.tensor_add
add_ip.write(0x10, x[0][0])
# Write other x values
add_ip.write(0x14, y[0][0])
# Write other y values
z[0][0] = add_ip.read(0x18)
# Read other z values

end_time = time.Time()

exec_time = end_time - start_time
print(exec_time)
```

Figure 10 - The proposed framework. A hybrid model as both CPU and FPGA are used. Note that even after the execution of the above commands, more commands can be executed which may or may not use an overlay.

Multiple executions of SampleRNN must be conducted to retrieve and tabulate the inputs and outputs of these commands to later compare with the hardware designed. The description of the commands along with their input and output results will form the basis to the following hardware design.

As mentioned earlier, there are four methods which can be used to implement hardware design on the FPGA. In this research, due to longer design-time required along with extremely high complexity, the VHDL approach will not be considered. Instead, the implementation uses High Level Synthesis (HLS) tools enabling hardware generation from C-like code. Between the three, the approach of using C Synthesis allows complex operations to be performed using broad available libraries, and inclusion of powerful compiler directives. Therefore, it is utilized here.

Ultimately, the IP is brought into Vivado 2018.2, a powerful software by Xilinx for FPGA designing and testing purposes. It begins with the block design, where other required elements (such as processors and interfaces) are connected to the IP designed above. Once the design is verified, synthesis and implementation phases are executed to finally generate a bitstream. This contains the binarized design information. Along with this, the physical block design must also be exported (as a .tcl file). Both these files will be needed to configure the overlay. Since the concept of an overlay is native, an application can include the hardware accelerator in between other lines of code with a few commands. Note that if multiple commands are to be accelerated, multiple overlays commands must be called for downloading bitstreams onto the FPGA or a single overlay can include multiple accelerators, which is subject to resources available.

3.1.4 Verification and Evaluation

Testing begins from the very early stages, where to select an application, multiple models are tested. Majority of testing will, however, be conducted during the design phases. The C synthesis program is debugged by exporting the programs as an RTL (Register-Transfer Level) within Vivado HLS to Vivado 2018.2. Once the bitstream is generated, the hardware can be provided inputs and the outputs can be compared using the mean-square error. Testing will ensure that the correct functionality has been accomplished within the hardware designed. The final stage of testing will be when the overlay is called within the SampleRNN program.

Evaluation includes obtaining performance metrics. This will answer whether the speed of the program has been accelerated, for which the original program must be timed while generating one second of audio. After embedding the accelerator into program, the timing performances must be recalculated. To do so in a Python program, the 'time' library can be used. Time checkpoints will be made before and after generating sound to obtain the execution time. A ratio representing the speed up factor can be calculated from this. Moreover, the same process can also be replicated on a high-end PC with/and without a GPU to illustrate the effectiveness of the new model. Speedup factor will be measured using statistical analysis over multiple executions.

The accuracy of the sound produced by the accelerated system will also be considered. This is particularly important in cases where hardware accelerators use different data representations than the original application. For this, the aim is to ensure that the overall differences cannot be perceived by the human auditory system (Nave, 2018). Ideally, it is expected that there are no differences in the sound samples when these are represented with as little as 16-bits. Accuracy issues are determined by FPGA not having predefined datatypes, unlike in Python, where the number of bytes can be increased to fit any number possible. In addition to accuracy, the speed performance of the accelerator is also affected by the datatype chosen. Since a fixed-point datatype will be utilized to provide high performances when compared to floating-point numbers, the results of mathematical calculations will definitely be degraded. On the other hand, the Mean Opinion Score (MOS) (ITU, 2016) can also be included as a measure to identify the depth with which quality is lost in perceived audio when using other datatypes.

3.2 Summary

The methodology to accelerate an RNN based sound synthesizer using an FPGA starts by first selecting a co-existing model. Followed by this, the program must be profiled to identify which operations within the network cause a significant delay. An in-depth analysis must be then conducted on these operations, eventually modelling prototype designs. These designs undergo multiple testing and refining. The end product of this will be a bitstream, which, upon uploading onto the FPGA, can be substituted in the model, where performances such as speed and accuracy are evaluated.

4 FRAMEWORK

This chapter details the proposed framework to develop ad-hoc accelerators for deep learning applications. At first, the development details and features of the framework are described. This is then followed by an example utilizing the framework to accelerate a simple application developed using the PyTorch library, showcasing the usability and performances of the framework.

4.1 Development

An illustration of the proposed framework is detailed in Figure 11. The framework consists of four layers, for which brief descriptions are provided below and further details, where needed, are outlined in the subsections to follow.

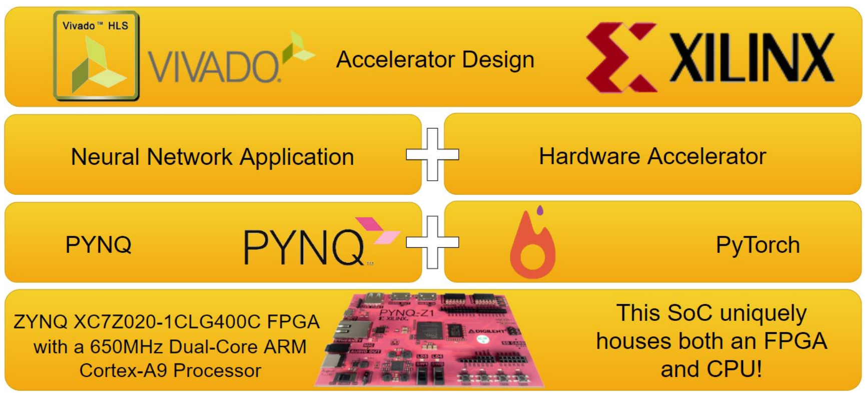

Vivado™ HLS
VIVADO
Accelerator Design
XILINX
Neural Network Application
Hardware Accelerator
PYNQ
PYNQ
PyTorch
ZYNQ XC7Z020-1CLG400C FPGA with a 650MHz Dual-Core ARM Cortex-A9 Processor
This SoC uniquely houses both an FPGA and CPU!

Figure 11 - Framework Structure.

Starting from the first layer, which indeed is the core of this system, is the Xilinx PYNQ-Z1 board (PYNQ, 2018). As mentioned in the previous chapter, the board consists of the ZYNQ-7000 SoC which houses a Dual-Core ARM Cortex-A9 processor alongside an FPGA, providing a combination of software and hardware programmability.

The second layer comprises of two software modules being combined for the first time ever. First of which is the PYNQ OS, which is a Debian-like Linux platform developed by Xilinx. It comes with many software packages which eases the development of applications which aim to use both, software and hardware together, unifying the two halves of the ZYNQ SoC. The second module within this layer is the NN development library for Python, PyTorch (2018), which was ported onto the PYNQ OS.

The first two layers will be packaged and provided as an image file to developers/the community. Whereas, layers three and four describe techniques to apply on top of the first two layers to accelerate an arbitrary NN application. In particular, the third layer consists of the specifics on how to exploit the available software packages from layer two to create a hybrid platform combining a PyTorch-developed NN application with a custom designed hardware accelerator.

Layer four contains the techniques to design the hardware accelerator used in layer three. This includes information on using the hardware development toolkits provided by Xilinx, namely, Vivado 2018.2 and Vivado HLS, described in the previous chapter. Although other methods to design hardware were proposed as well, using Vivado HLS to develop C synthesis programs was deemed the most appropriate within the context of this framework.

4.1.1 Environment Setup

To be able to use the PYNQ-Z1 board, the PYNQ OS v2.3 image (PYNQ, 2018) is flashed onto a MicroSD card and placed into the slot on-board. Proceeding forward, simply using a web browser to access http://pynq:9090, while being connected to the same network, will open the standard Jupyter notebook interface. Here, new python scripts can be developed and tested quickly. Once the access to the board is established through an SSH client, the steps devised to port PyTorch, together with a test and its result to verify whether this was successful can be found in Appendix C.

4.1.2 Accelerator Design

Assuming that the critical kernels within the program are known before proceeding into the design of hardware, some details with respect to the inputs, outputs, and processing are still ambiguous. Hence, the following are aspects that need to be studied (for each critical kernel):

1. Inputs:
    a. Knowing the source of the inputs will later provide guidance on where the hardware should be invoked from. This includes performing any pre-processing codes required. More information on this in the next subsection.
    b. The source will also help in determining the data structure, for example, if the input is a scalar or an array. If the data is an array, an extension to this would be to know what the dimensions are. This must be consistent with how the hardware will be designed.

    c. Additionally, the input datatype can also be procured, either by understanding the how the input was generated by the program or by monitoring the values. For example, if the data is coming from the output of a function which produces values which have a range from negative one to positive one, the hardware can be optimized for this.

2. Processing:
    a. The information on what needs to be done can easily be obtained through the documentation, as descriptions of what a particular function is meant to perform are usually mentioned. The goal of the accelerator designed will be to match this description. Knowing what the processing is can also decide what datatype must the output of the function be.
3. Outputs:
    a. Once the accelerator has completed the computations, it must be placed appropriately so that the application can continue as it would normally. Furthermore, the location of the post-processing of data must also be conducted right before this location.

Once all information about a critical kernel has been obtained, developing a function in software, which mimics its functionality with low-level python commands, will further increase the knowledge about the kernel and allow an easier process of porting it into hardware. It further verifies whether or not the technical information is correct.

Moving on, the hardware can now finally be designed. This process can be broken-down into two segments: first is where the hardware which mimics the targeted kernel is designed, and the second is where that hardware, exported as an IP, will be placed along with other hardware blocks which will help in the communication between the CPU and FPGA.

For the first segment, as mentioned earlier, Vivado HLS was chosen as the platform to develop the hardware. This was due to System Generator not having the capability to communicate large sums of data in and out of the design, which is typically seen here. Using Vivado HLS involves writing C synthesis code which is translated into hardware. There are many techniques that are available to improve the performance of the program which are detailed below, however, before doing so, the first step here would be to port the python function written earlier to mimic the kernel into a C synthesis program inside the Vivado HLS tool.

When using the tool, a new project must be created. Here, a Top Function name must be provided along with a C synthesis file which will contain that function. When invoking this hardware later, this function will be the one to be executed. A tip here would be to avoid using the name 'main' for the function, as it confuses the synthesizer into thinking that it is the main often seen in a C++ program. The drawback of this is that the main, when provided inputs, must be in the argument count and argument vector format, which is not possible here. Moving on, for the Solution Configuration, the Part Selection will have to be changed into XC7Z020CLG400-1, which selects the FPGA chip available on the PYNQ-Z1 board.

The inputs and outputs of the hardware accelerator must be the input of the Top Function. Two of the ways data can be exchanged from the accelerator involve using either the AXI4-Stream or the AXI4-Lite interfaces. The latter is used when a single transaction is needed at a time. This includes passing the address of the element each time the data is exchanged. The former, on the other hand, does not have the concept of an address other than the base (Xilinx, 2011) and uses the DMA. Here, large arrays of 32- or 64-bit data can be exchanged at far better bandwidths than the AXI4-Lite since there is no need to continuously provide addresses and transactions are one-shot. The transaction must be either of 32- or 64-bit elements is because of the settings on Vivado, shown later, which allows only the stated widths going on the bus through the DMA. Although, when using small sums of data, the AXI4-Lite will provide better results as there will be no need to include the DMA overheads.

In terms of programming, when using the AXI4-Lite, Vivado HLS provides information of the predefined locations where the accelerator will be reading data from. Therefore, the Python program will have to simply write/read those addresses. When streaming data, on the other hand, since the DMA is directly connected to the accelerator, the program will simply pass contiguous arrays. In reality, the contiguous arrays are already on the DDR and so the DMA actually passes the base address and the number of bytes only to the accelerator. Whichever technique is used, to be able to exchange data from a Python program, an AXI interface must be deployed.

When it comes to designing hardware, resources and performance are often parameters discussed. To enhance these parameters, designers take advantage of the capability of hardware to have selective fixed-point datatypes at signal-level. In Vivado HLS, on the other hand, the concept is extended to variable-level. This implies that individual variables can have unique fixed-point datatypes using libraries provided by Xilinx. As mentioned earlier, for datatype optimizations, knowing the dynamic range of the data is vital. Vivado HLS provides developers the option of using either floating-point or fixed-point datatypes within the design. The former indeed provides high accuracy results with full-precision computations and eliminates spending valuable development time in investigating the dynamicity of data. However, the latter comes with a wide range of advantages, these include, reducing the amount of logic resources used, reducing the power consumed, and increasing the execution speed of the hardware (reducing latency) (Finnerty and Ratigner, 2017). By default, Vivado HLS uses floating-point. But, with the inclusion of specific libraries demonstrated in the example later, fixed-point datatype can be introduced within the design, with any configuration, which implies that the developer is given the capability to choose how many bits to allocate for the integer and fractional parts respectively.

It may seem that the C synthesis program, which natively is sequential programming, will provide hardware being executed sequentially as well. This, however, is not the case within Vivado HLS, where the synthesizer is capable to identify segments which can be executed in parallel and so applies the optimizations. Different pragmas can be used to activate optimizations within the hardware. These can be found, with examples on how to use them, in the Xilinx (2019) documentation.

4.1.3 Generating Bitstream

Now when the C synthesis program has been developed and the optimizations have been applied, the design must be synthesized and exported to Vivado 2018.2 as an Intellectual Property (IP). To do so, the icons shown in Figure 12 are clicked.

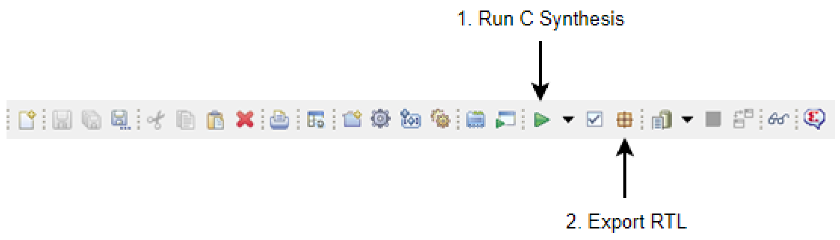


Figure 12 - Steps to export the designed hardware from Vivado HLS to Vivado 2018.2 as an IP.

Upon clicking on the Export RTL icon, a pop-up window will provide some configuration options. The Format is set as IP Catalog whereas, the language is VHDL instead of Verilog. All boxes should be left unchecked.

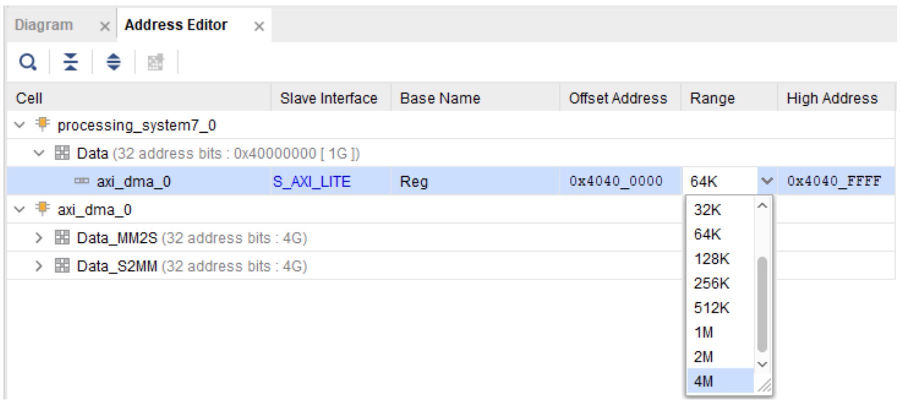
Next, a new project is created on Vivado 2018.2. Before doing so, the PYNQ-Z1 board files are added to the Vivado folder. While creating the project, the PYNQ-Z1 board should now be visible. Next, the repository where the IP was generated should be added to the project so that it is available to be utilized.

Moving on, a new block design must be created. The first component to add is the ZYNQ7 Processing System, along with this, the block automation can also be executed. The next block to be added is the accelerator exported from Vivado HLS. Simply searching the Top Function name chosen before will show the IP, if not, the repository has not been added correctly. Subsequently, the AXI DMA block should also be added. Before connection automation is done, configure the DMA block, by double-clicking it, to configure it:

- Uncheck the Scatter Gather Engine.
- Increase the width of buffer length register from 14 bits to 26 bits. This is changed to allow more data to be transferred in a single transaction (from 2^{14} = 16KB to 2^{26} = 64MB).

Now, the input of the accelerator IP should be connected manually to the M_AXIS_MM2S port of the DMA. The output of the accelerator should be connected to the S_AXIS_S2MM port of the DMA. Thereafter, add the high-performance slave on the ZYNQ7 Processing System, again by double-clicking it, and click on the PS-PL Configuration page. At this point, extend the HP Slave AXI Interface and enable the HP0 interface. Run Connection Automation twice.

Before generating the bitstream, using the Address Editor, the range of Data accessed by the ZYNQ processor must be increased from 64K to 4M. This is shown in Figure 13 below.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
Data_S2MM (32 address bits : 4G)					

Figure 13 - Increasing the total memory range of the ZYNQ on Vivado 2018.2.

Using the Regenerate Layout and Optimizing Route options available in block design, the connections and placement of blocks will be as efficient as possible. The block design should now look like the one shown in Appendix D. Before the bitstream can be generated, from Sources, an HDL wrapper for the block design should be created (right-click on the correct block design name). The hardware then must also be exported as a .tcl file (File -> Export -> Export Block Design).

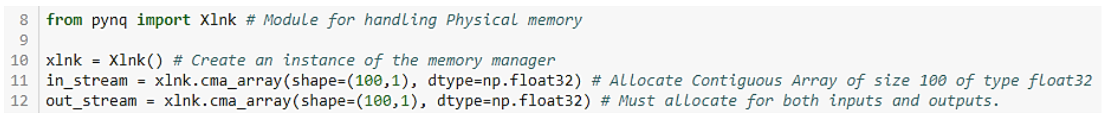
4.1.4 Embedding an Accelerator

Now that the .bit and .tcl files are generated, these should be placed onto the PYNQ-Z1 board into the same folder as the main python program which will be executed. Note that these two files (.bit and .tcl) must be the same name. To include the accelerator within a python program, the Overlay library will be used. In addition, the DMA block in hardware can also be referenced. An example of this is shown in the Figure 14 below.

```python
import torch
import pynq.lib.dma # DMA Package
from pynq import Overlay # Overlay Package

overlay = Overlay('./nameOfBitstreamFile.bit', download=True)
dma1 = overlay.axi_dma_0 # Verify with Block Design in Vivado 2018.2 For name of DMA block
```

Figure 14 - Using the Overlay library to introduce the designed hardware within the python program.

If multiple overlays are used, an additional download input parameter can be specified (shown in Figure 14), True or False, which can control whether the overlay bitstream will be downloaded onto the FPGA or not. Next, since the DMA is being used to transfer data, physical memory needs to be allocated using another module provided in the PYNQ library shown in Figure 15.



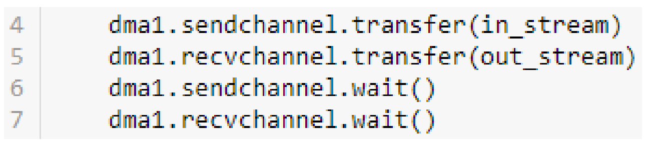
```python
from pynq import Xlnk # Module for handling Physical memory

xlnk = Xlnk() # Create an instance of the memory manager
in_stream = xlnk.cma_array(shape=(100,1), dtype=np.float32) # Allocate Contiguous Array of size 100 of type float32
out_stream = xlnk.cma_array(shape=(100,1), dtype=np.float32) # Must allocate for both inputs and outputs.
```

Figure 15 - Allocating memory. Done for both input and output of the accelerator.

Once the allocation is complete, the accelerator can finally be provided input data and the output data can be read once the computation is complete. However, when using the accelerator, the data being passed can come from various sources. In some cases, the I/O might be of different sizes and so it might be useful to apply some pre-/post-processing. Usually, the I/O is kept as row vectors. Furthermore, data passed to the DMA must a numpy array (NumPy, 2019).

Some useful functions for handling this include, concatenation (torch.cat()), reshaping (torch.reshape()), obtaining data values only (tensorArray.data) functions which can be called on tensors. To convert from a tensor to a numpy array or vice versa, the tensorArray.to_numpy() and torch.tensor(numpyArray) can be used. More details for these can be found in the documentation.

When the data is converted into a numpy array, it can then be transferred to the accelerator using the DMA. To do so, the code shown in Figure 16 should be used. When the receiving channel returns from the function call and the program proceeds, the result will be found in the out_stream array.



```python
    dma1.sendchannel.transfer(in_stream)
    dma1.recvchannel.transfer(out_stream)
    dma1.sendchannel.wait()
    dma1.recvchannel.wait()
```

Figure 16 - Transferring data to the accelerator and waiting for the output to be complete.

4.2 Utilizing the Framework

As an example, to showcase the usability and performances of the proposed framework, this section aims to accelerate the basic RNN layer within PyTorch. The program developed instantiates the RNN layer with 128 input features, 128 hidden features, and two layers. When an RNN has more than one layer, it can be thought to have multiple RNN stacked together. An illustration of this is shown in Figure 17. Then, the program generates two random-valued tensors as inputs to the network, one of which is the actual input, whereas the other is the initial hidden state (labelled hidden states in the figure below). On top of these two inputs, the accelerator will also have to be provided with the weights and biases of the network. The weights, in particular, are the largest arrays within this context.

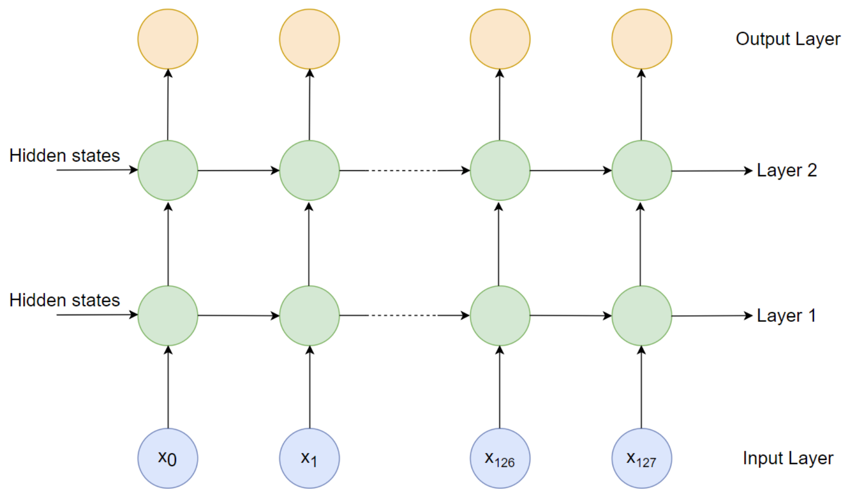


Figure 17 - The RNN layer in PyTorch. As seen, there are two hidden layers, 128 input and hidden features.

For each hidden layer, the following equation must be calculated (taken from the PyTorch Documentation (2019)):

$$\boldsymbol{h_t} = \tanh(w_{ih}x_t + b_{ih} + w_{hh}h_{(t-1)} + b_{hh}) \qquad (2)$$

Here, w_{ih} and w_{hh} are 128x128 matrices. Whereas, x_t and $h_{(t-1)}$ are three-dimensional tensors of size, 1x1x128 and 2x1x128 respectively. The biases, b_{ih} and b_{hh}, are of size 128x1.

The final output of the network will be the output of the second hidden layer. Note that the input to the second hidden layer will be the output of the first. In addition, another output of the RNN layer forward pass function call is a combination of both the outputs from the hidden layers. Indeed, the actual output of the network can be found within this tensor as well.

The understanding of the inputs, processing, and outputs was straight-forward in this case. Since the only computational kernel within the program is the RNN layer, no further profiling was conducted. Advancing forward, the functionality was mimicked in software, for which the code can be found in Appendix E. In order to verify the results, the mean-square error of the output tensors was taken. This gave an error of zero. This is expected as the function uses the same functions in PyTorch as the original RNN layer would.

Moving on, since the inputs to the accelerator were of different sizes, pre-processing is needed. On top of this, since the accelerator is going to be performing matrix multiplication, the process of moving from a three-dimensional matrix to a row vector had to be considered carefully. For simplicity sake, if a 4x4 matrix was to be converted to a row vector, the transform used is shown in Figure 18, where the values represent the positions they hold within the row vector.

$$\begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

Figure 18 - Converting a matrix into a row vector. The value at each position represents what position the actual value of the matrix would take in the row vector.

Now that it is known what the inputs are, how are they going to be fed into the accelerator, what must be performed on them, and where they must be returned, the Vivado HLS design is coded. This program can be found in Appendix F. The generic Vivado 2018.2 block design is followed for this accelerator as well (Appendix D).

Appendix G consists of a program and results which compare the performance of the CPU against the accelerator. The program is designed such that the accelerator and the CPU are provided the same inputs and so there is no need to keep track of a couple of inputs and their respective outputs now (done for quick debugging). The speedup factor achieved here is about 21 times faster execution (excluding time taken for pre/post processing) while the mean-square error was about $2.5887\text{x}10^{-6}$. Although the error has increased due to the introduction of fixed-point datatype, it is still minuscule.

The utilization of the designed accelerator is shown in Table 1. These results indicate that the FPGA chip still has reasonable amount of space for other hardware to be place as well. The program loops were not able to be further unrolled as this would not fit the FPGA chip.

Table 1 - Utilization statistics from Vivado 2018.2.

LUT %	FF %	BRAM %	DSP %	LUTRAM %
37.47	18.87	21.79	41.36	6.58

5 ACCELERATING SAMPLERNN

The following chapter discusses the acceleration of SampleRNN using the proposed framework. The results of profiling and the analysis of kernels are described here. Then, the kernels are re-implemented in software. Thereon, details on the Vivado HLS and Vivado 2018.2 designs are given, as well as the information of the accelerator integrated into the program to test the overall performance is provided.

5.1 Dependencies

To be able to execute the SampleRNN program, the PYNQ OS is still missing some packages after porting PyTorch. This is the LibROSA (2018) python package, which handles music and audio analysis. Yet again this package had to be ported onto this platform. The step-by-step guide along with a small test made can be found in Appendix H. Next, the SampleRNN source code can be cloned from the git repository (Kozakowski et al., 2017) and executed.

When executing the SampleRNN application with the configurations recommended, it was noticed to have been using the SWAP when the on-board RAM was falling short. This led to an analysis of what dimension size, which refers to the number of neurons within each layer, should be set to avoid the application using the SWAP. Indeed, using the SWAP will further slowdown the application, leading to an unfair comparison with the accelerator. The `top` Linux command was used in this case to monitor statistics referring to the percentage of SWAP used by all processes running. As a result of doing so, the dimension found was 256 instead of the default 1024. However, when designing the accelerators, due the inputs being large and therefore not fitting the FPGA chip, the dimension was eventually set to 64. Hence, the current performance of the application is shown in Table 2 below. The specification of the machine it was executed on are: Quad-core Intel(R) Core™ i7-6500U CPU at 2.50GHz-2.60GHz (overclocked up to 3.2GHz), 16GB RAM, 4GB NVIDIA GeForce GTX 960M, running 64-bit Ubuntu 18.04LTS. Along with this, the table also consists of the performance on the PYNQ-Z1 board with the said dimensions.

Table 2 - Current performance of SampleRNN with dimension set as 64.

Platform	Number of Samples Generated	Time Taken (secs)	Output Audio Rate (Hz)
CPU	16,000	90.77	176.26
CPU + GPU	16,000	23.13	691.61
PYNQ-Z1	16,000	1798.74	8.90

5.2 Profiling

As mentioned in the methodology chapter, two levels of profiling will be conducted. First of which will narrow down the search from program-level to function-level, whereas the second will provide information from function-level down to line-level.

5.2.1 Top-level Profiling

To conduct the top-level profiling, the program was executed with a sample length of 16,000 (one second of audio), dimension of 64 and the rest of the default settings. The profiling results can be found in Appendix I. The kernels which occupy the most amount of time have also been highlighted in yellow within the results. These have to be further investigated at line-level to identify the root of the bottlenecks. Some kernels have not been considered since they are processes within Python which carry out operations which will be infeasible in hardware. On the other hand, some kernels either invoke others or are out of the scope of this research and hence are also not considered.

Within the results, the parameter *ncalls* represents the number of times a function is called. Sometimes there are two numbers mentioned here, the first of which is the number of times invoked whereas the second is how many times the invocations were non-recursive calls. The *tottime* suggests the time spent within a certain function excluding the sub-functions called. The first *percall* column is simply a division of the *ncalls* and *tottime*. c*umtime* represents the time elapsed from invocation till exit, whereas the *percall* followed by this is the division between *cumtime* and *ncalls*. The last column, labelled *filename*, provides file location information on the function which is responsible for the respective statistics shown in that row.

5.2.2 In-depth Profiling

The location of the highlighted functions can be taken from the last column of the top-level profiling in Appendix I. The line profiler developed by Kern (2019) was additionally installed. The results of line profiling can be found in Appendix J.

5.3 Identifying Bottlenecks and their Analysis

From the results of profiling, it can be deduced that there are multiple kernels which can be accelerated. Although most of the results seen from both profiling seem to have functions calling other functions, where those are also seen in the results, this can be misleading. After identifying such cases, it was found that the two kernels that could be ported into hardware are, the GRU matrix multiplications and vector additions, and weight normalisation calculations. Both are further explained below.

5.3.1 Gated Recurrent Unit (GRU)

As mentioned in the Literature Review chapter, GRU refers to mechanisms which deal with the gradient problems within an RNN. GRUs can selectively store dependencies from different time instances although it does not contain memory cells but, instead has variables which store the past hidden states. They are far less complicated in design and outperform LSTM cells in certain applications (Chung et al., 2014). The following equations have been taken from the PyTorch GRUCell Documentation (2019) which describe this kernel:

$$\mathbf{r} = sigmoid(\mathbf{W_{ir}X} + \mathbf{b_{ir}} + \mathbf{W_{hr}H} + \mathbf{b_{hr}}) \tag{3}$$

$$\mathbf{z} = sigmoid(\mathbf{W_{iz}X} + \mathbf{b_{iz}} + \mathbf{W_{hz}H} + \mathbf{b_{hz}}) \tag{4}$$

$$\mathbf{n} = \tanh(\mathbf{W_{in}X} + \mathbf{b_{in}} + \mathbf{r} * (\mathbf{W_{hn}H} + \mathbf{b_{hn}})) \tag{5}$$

$$\mathbf{h'} = (1 - \mathbf{z}) * \mathbf{n} + \mathbf{z} * \mathbf{h} \tag{6}$$

The variable **r** here refers to the reset gate calculation. This decides whether or not to store the current hidden state and if any of the already stored states should be removed. Variable **z** decides how much of the current hidden state be passed to the next hidden state. Variable **n** is the memory variable. Finally, variable $\mathbf{h'}$ is the new hidden state generated based on the previous hidden state input and the new memory variable, with variable **z** deciding to what extent should this be done (Mohammadi et al., n.d.).

The line profiling results which refer to this kernel are the functions GRUCell, and linear. To generate a second of audio at 16 kHz, these equations had to be computed 2500 times. For the matrix and vector dimensions mentioned below, this meant that, in total, 61 million multiplications and 62 million additions had to be performed. For audio at 48 kHz, these values would be three times larger, clearly showing the number-crunching involved. As seen in the profiling results for GRUCell, the implementation of the above equations is shown, where about 84.6% of the time spent in the linear function. This function performs the matrix multiplication (weights with the input and the hidden state, respectively) and the vector addition (biases). Meanwhile the CPU can very well handle the calculation of the non-linear functions (tanh and sigmoid) and variable $\mathbf{h}'$.

Henceforth, the focus of this accelerator will be similar to the RNN layers from the previous section. However, in this case, the matrix multiplication will be on matrices of size 192x64 (weights) and 64x1 (inputs and hidden state, each). The vector addition will be between the output of the matrix multiplication, 192x1, and the bias, which, of course, is also 192x1. The function which mimics this behaviour was not needed as the RNN layers verified the functionality earlier.

As for the inputs, these can be fetched from the GRUCell function itself. The output can be placed into variables called, *gi* and *gh* (shown in the line profiling). Doing so will allow the program to continue with execution as it would normally. On the downside, there does not seem to be a way to have multiple invocations of this function so that multiple hardware blocks can be placed to have parallel execution of this kernel. This is due to the fact that this kernel is not explicitly called and nor is there a way to continue execution without having the result.

5.3.2 Weight Normalisation

Weight normalisation, as the name suggests, normalizes the weights within the NN, by decoupling the weight magnitude (*g*) from the direction (*v*). The equation which describes this behaviour, taken from the PyTorch Documentation (2018b), is shown here:

$$\mathbf{W} = \mathbf{g}\frac{\mathbf{v}}{||\mathbf{v}||} \tag{7}$$

For three-dimensional tensors, the magnitude/norm is calculated per layer. For instance, if the tensor is of size 64x64x1, the magnitude will be the square-root of the sum of the squared 64 elements in each layer. This will result in a 64x1x1 tensor which will be multiplied, element-wise, with *g*, which must also be 64x1x1.

This computational kernel occupies up to 68% of the execution time as seen in the top-level profiling results and proves to be a critical aspect within SampleRNN. The functions which represent this kernel are compute_weight, and _norm. The inputs, *g* and *v*, both can be found within the compute_weight function and the output, *w*, is the return parameter of the function. The sizes of the for the inputs are slightly more complicated than seen previously. Input dimensionality can vary significantly, as shown in Table 3, which indicates that some dimensions barely occur, and thus these are not considered for hardware implementation. In addition, the accelerator for dimension 64x256x16 do not fit on the FPGA chip. Therefore, hardware is only designed for dimensions, 64x64x1 and 256x64x1, which is compliant with Amdahl's Law by making the common/frequent case faster. All in all, three million square-roots, three million divisions, 330 million multiplications, and 165 million additions, on average, had to be conducted to generate one second of audio at 16,000Hz.

The python program which mimics the functionality is shown in Appendix K. This implementation is for any arbitrary size.

Table 3 - The different dimensions the input *v* and *g* can take.

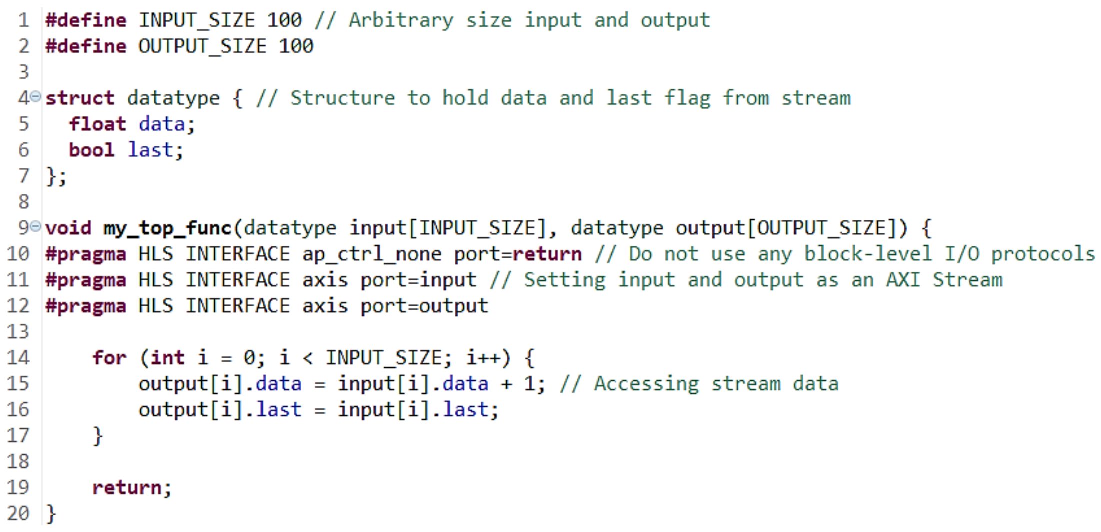
Size of *v*	Size of *g*	Occurrences in Percentage
64x16x1	64x1x1	1.98%
64x64x16	64x1x1	1.98%
64x64x1	64x1x1	32.18%
64x64x4	64x1x1	0.50%
64x256x16	64x1x1	31.68%
256x64x1	256x1x1	31.68%

5.4 Hardware Design & Integration

Now that all the information about what needs to be ported into hardware has been accumulated, the accelerators are designed in Vivado accordingly. The subsections below outline design practices that were followed. And then finally the accelerator specifications are mentioned.

5.4.1 *Using the AXI Stream*

Since large sums of data will be transmitted, the AXI Stream interface is the most efficient way to do so for reasons mentioned previously. To configure, and read/write data using this, an example is provided in Figure 19 below:



```
 1 #define INPUT_SIZE 100 // Arbitrary size input and output
 2 #define OUTPUT_SIZE 100
 3
 4 struct datatype { // Structure to hold data and last flag from stream
 5   float data;
 6   bool last;
 7 };
 8
 9 void my_top_func(datatype input[INPUT_SIZE], datatype output[OUTPUT_SIZE]) {
10 #pragma HLS INTERFACE ap_ctrl_none port=return // Do not use any block-level I/O protocols
11 #pragma HLS INTERFACE axis port=input // Setting input and output as an AXI Stream
12 #pragma HLS INTERFACE axis port=output
13
14     for (int i = 0; i < INPUT_SIZE; i++) {
15         output[i].data = input[i].data + 1; // Accessing stream data
16         output[i].last = input[i].last;
17     }
18
19     return;
20 }
```

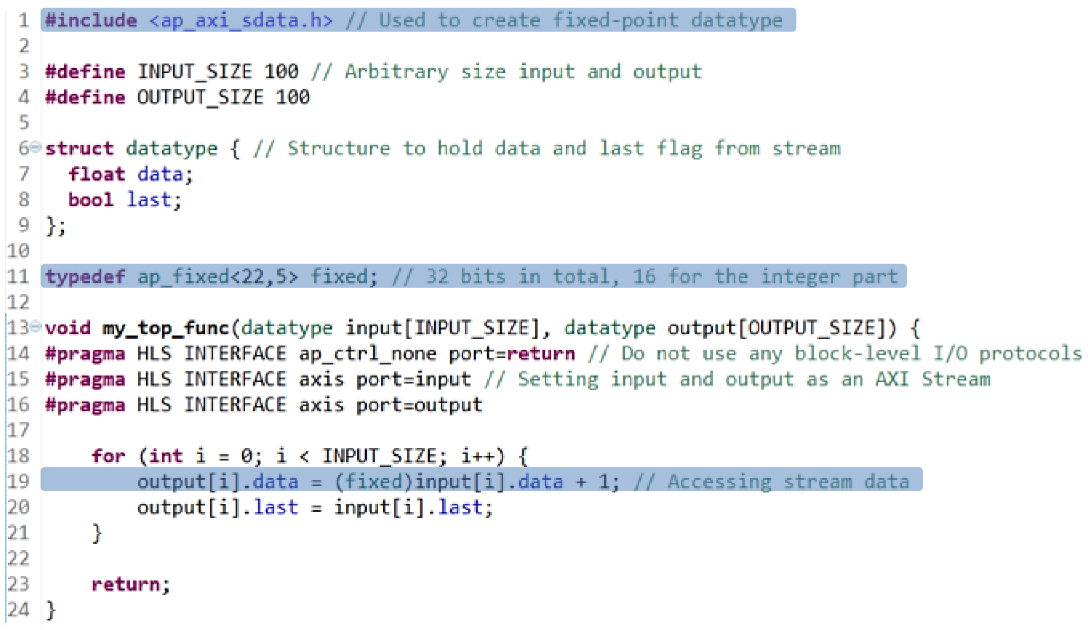
Figure 19 - AXI Stream example on Vivado HLS.

As seen in the figure above, the AXI Stream items come into the program as a structure of type *datatype*. This contains a data element, of type float, and a flag which indicates whether this item is the last or not. The default structure used also contains other signals, however, using the above two can provide all functionality needed. Next, in order to inform the compiler that the I/O are a stream, the pragmas seen in lines 10-12 are used. Once this is complete, the data can now be accessed within the body of the function. In the example above, the output is the input incremented by one.

In addition, the structure, *datatype*, does not necessarily need to have float but, it must either be 32-bit or 64-bit wide standard C/C++ types. Another rule when using AXI Stream is that the stream cannot be read/written out of order. Further, writing must also be done within a loop which consists of only writing out data, else it is taken as an irregular pattern, which is prohibited.

5.4.2 *Fixed-point Datatype*

Currently, the example shown in Figure 19 uses floating-point datatype throughout the program, which is considered having FIX_32_23 (IEEE, 2008). Knowing the dynamic ranges of the I/O here becomes essential to eliminate any extras being used. For instance, if the input has values ranging from -10 to 10, assigning 8 bits for the exponent is inefficient. Instead, providing five bits will give the same functionality. Although, for such a simple example, performances are not greatly improved, in the actual accelerators designed for SampleRNN, employing fixed-point datatype makes the hardware up to twice as fast as it opens a gateway to a new set of specialised libraries dedicated for fixed-point operations. An example of how fixed-point datatypes can be used within the previous example is shown in Figure 20 below:



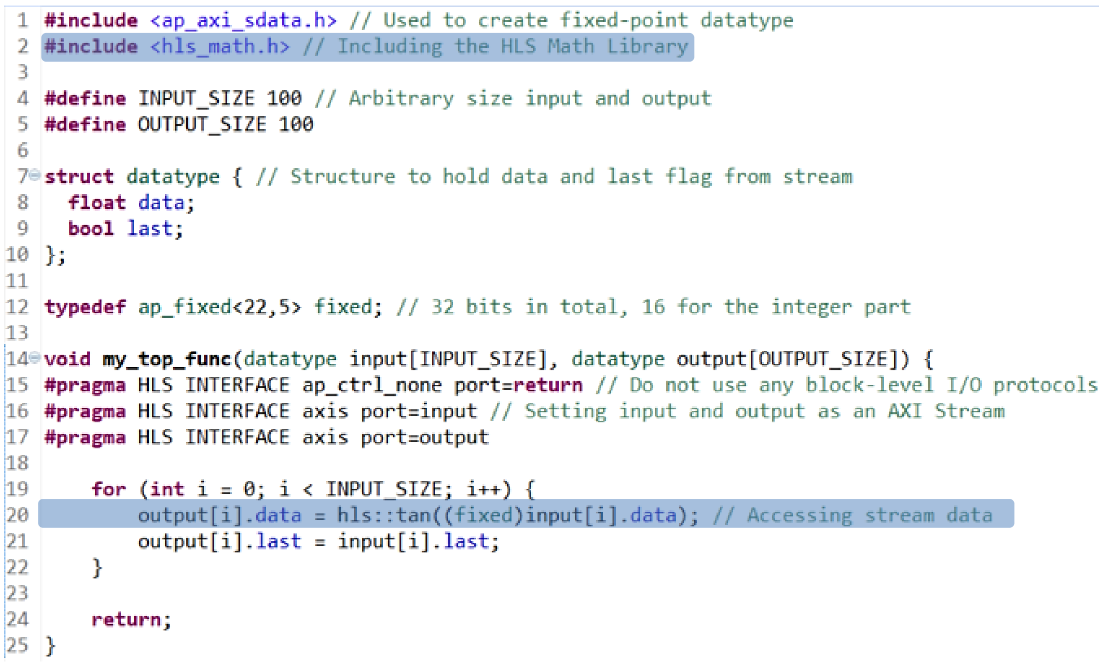
```
#include <ap_axi_sdata.h> // Used to create fixed-point datatype

#define INPUT_SIZE 100 // Arbitrary size input and output
#define OUTPUT_SIZE 100

struct datatype { // Structure to hold data and last flag from stream
  float data;
  bool last;
};

typedef ap_fixed<22,5> fixed; // 32 bits in total, 16 for the integer part

void my_top_func(datatype input[INPUT_SIZE], datatype output[OUTPUT_SIZE]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // Do not use any block-level I/O protocols
#pragma HLS INTERFACE axis port=input // Setting input and output as an AXI Stream
#pragma HLS INTERFACE axis port=output

    for (int i = 0; i < INPUT_SIZE; i++) {
        output[i].data = (fixed)input[i].data + 1; // Accessing stream data
        output[i].last = input[i].last;
    }

    return;
}
```

Figure 20 - Inclusion of fixed-point datatype in Vivado HLS. Highlighted in blue are the changes from the previous snapshot.

The ap_axi_sdata.h library is already available within the Vivado HLS environment. To use it, a new type must be defined, this must be either ap_fixed<W, I> for signed or ap_ufixed<W, I> for unsigned, where W is the total width of the type, and I is the is number of bits from the total used for the exponent part. There are no limitations as to how many datatypes can be created, and so the design can incorporate multiple types and simply cast the signals to convert it.

5.4.3 Enhancing Computations

Now that the data is flowing through the AXI Stream, and the datatype has been casted to fixed-point, it is time to implement the computational kernel in hardware. To further speedup the kernels, the hls_math.h library can be utilized when dealing with fixed-point mathematical operations. A full list of available functions can be found in the Xilinx UG902 (2018c, p. 902) documentation, under section 'The HLS Math Library'. An example of this is shown in the Figure 21 below.



```
#include <ap_axi_sdata.h> // Used to create fixed-point datatype
#include <hls_math.h> // Including the HLS Math Library

#define INPUT_SIZE 100 // Arbitrary size input and output
#define OUTPUT_SIZE 100

struct datatype { // Structure to hold data and last flag from stream
  float data;
  bool last;
};

typedef ap_fixed<22,5> fixed; // 32 bits in total, 16 for the integer part

void my_top_func(datatype input[INPUT_SIZE], datatype output[OUTPUT_SIZE]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // Do not use any block-level I/O protocols
#pragma HLS INTERFACE axis port=input // Setting input and output as an AXI Stream
#pragma HLS INTERFACE axis port=output

    for (int i = 0; i < INPUT_SIZE; i++) {
        output[i].data = hls::tan((fixed)input[i].data); // Accessing stream data
        output[i].last = input[i].last;
    }

    return;
}
```

Figure 21 - Highlighted in blue are changes made to include the tangent mathematical function.

In particular, the two most useful directives for any type of accelerator is the pipeline and unroll directive, which is only used within loops of a program.

Pipelining is a technique which aims to increase parallelism by beginning execution of another instruction before the previously issued one is completed. Figure 22 illustrates the difference between a loop which has not been pipelined and one which has with Initiation Interval (II) equal to one. The II refers to the number of clock cycles that must kept between instructions being issued.

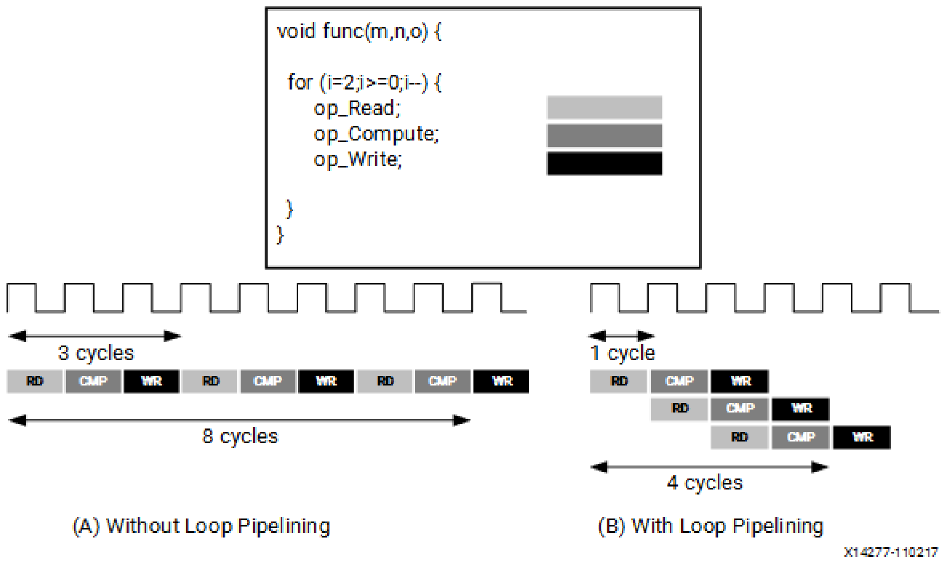


Figure 22 - (A) Illustrates the program at the top of the figure being executed sequential like any other C/C++ program loop. (B) The very same program is shown but now with the pipeline directive being used with the initiation interval equal to one. (Figure taken from Xilinx documentation (2019b).

For the example in Figure 22, if the II was set as four, the program would be executing sequentially. An example of the II equal to two is shown in the Figure 23 below. The larger the II value, the more it moves towards sequential execution.

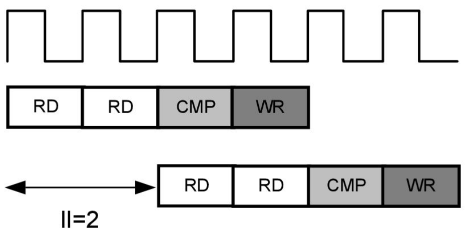
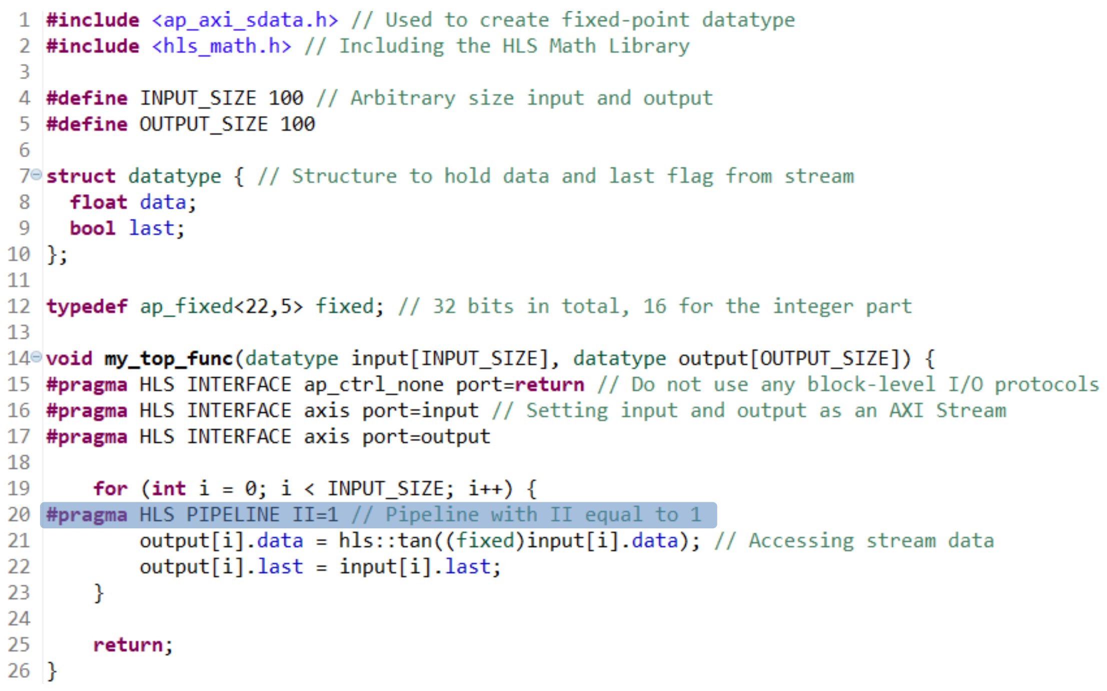


Figure 23 - Pipeline with II equal to 2. Taken from Xilinx (2019b)

Figure 24 demonstrates how to incorporate the pipeline directive in Vivado HLS. When included, a speedup factor of about just over four times was achieved when compared to without using the directive within this simple example.



```
#include <ap_axi_sdata.h> // Used to create fixed-point datatype
#include <hls_math.h> // Including the HLS Math Library

#define INPUT_SIZE 100 // Arbitrary size input and output
#define OUTPUT_SIZE 100

struct datatype { // Structure to hold data and last flag from stream
  float data;
  bool last;
};

typedef ap_fixed<22,5> fixed; // 32 bits in total, 16 for the integer part

void my_top_func(datatype input[INPUT_SIZE], datatype output[OUTPUT_SIZE]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // Do not use any block-level I/O protocols
#pragma HLS INTERFACE axis port=input // Setting input and output as an AXI Stream
#pragma HLS INTERFACE axis port=output

    for (int i = 0; i < INPUT_SIZE; i++) {
#pragma HLS PIPELINE II=1 // Pipeline with II equal to 1
        output[i].data = hls::tan((fixed)input[i].data); // Accessing stream data
        output[i].last = input[i].last;
    }

    return;
}
```

Figure 24 - Pipelining in Vivado HLS.

On the other hand, unrolling is another technique where parallelism is taken to a greater extent than seen in pipelining. In pipelining, the II cannot be less than one. Unrolling is equivalent to pipelining but with the II set to zero, which implies that all instructions within a loop will be executed in parallel. The concept can be visualized in Figure 25.

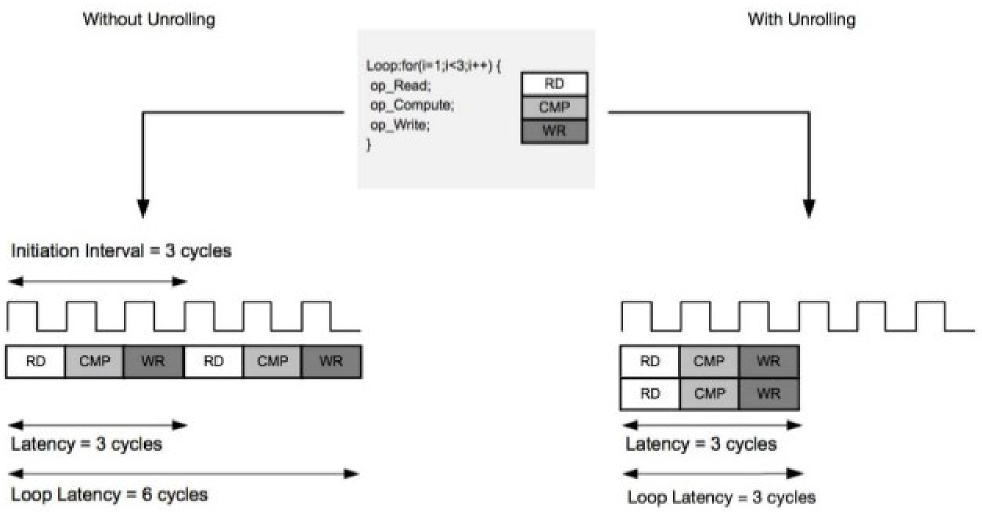
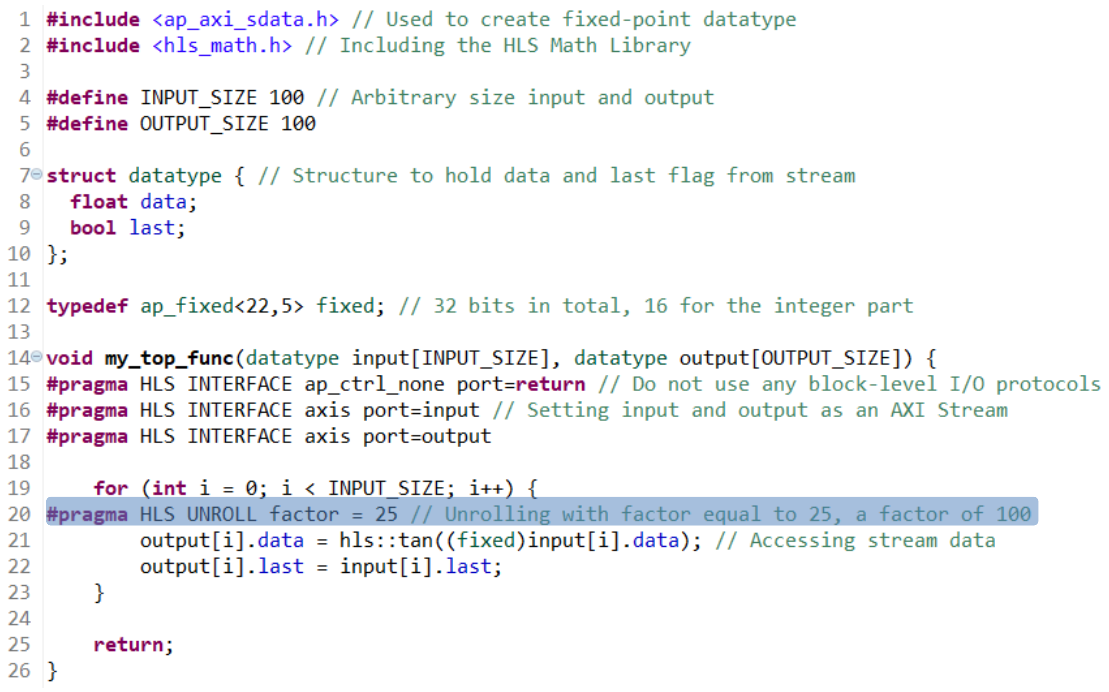


Figure 25 - The difference between executing a program with and without unrolling. Figure taken from NECST Lab @ Politecnico di Milano (2018)

The unrolling factor refers to the number of times the loop will be unrolled. The compiler, when provided this directive, copy-pastes the body of the loop a certain amount of times (factor) and then execute all those instructions in parallel. Therefore, unrolling with a large factor can lead to a situation where there is no more space available on the FPGA. This is due to the multiple resources needed to execute instructions in parallel unlike in pipelining where the resources are shared.

The unrolling directive can be utilized within Vivado HLS as shown in Figure 26. The unrolling factor must be a factor of the total number of times the loop will execute. Although Vivado HLS will allow numbers which are not factors, there are no advantages of doing so, as it means that towards the end of the loop execution, there will be empty slots which provide no difference to performances. If the unroll factor is not provided, the synthesizer unrolls the loop completely.



```
#include <ap_axi_sdata.h> // Used to create fixed-point datatype
#include <hls_math.h> // Including the HLS Math Library

#define INPUT_SIZE 100 // Arbitrary size input and output
#define OUTPUT_SIZE 100

struct datatype { // Structure to hold data and last flag from stream
  float data;
  bool last;
};

typedef ap_fixed<22,5> fixed; // 32 bits in total, 16 for the integer part

void my_top_func(datatype input[INPUT_SIZE], datatype output[OUTPUT_SIZE]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // Do not use any block-level I/O protocols
#pragma HLS INTERFACE axis port=input // Setting input and output as an AXI Stream
#pragma HLS INTERFACE axis port=output

    for (int i = 0; i < INPUT_SIZE; i++) {
#pragma HLS UNROLL factor = 25 // Unrolling with factor equal to 25, a factor of 100
        output[i].data = hls::tan((fixed)input[i].data); // Accessing stream data
        output[i].last = input[i].last;
    }

    return;
}
```

Figure 26 - Unrolling in Vivado HLS.

In some scenarios, however, these directives may fail to be applied by the synthesizer. This usually occurs for a couple of reasons. First of which could be due to dependencies, either among instructions within the body of the loop or loop-carried. Or there might be a scheduling problem, which may occur due to resources being utilized by another command (seen in pipelining). In some cases, common to unrolling, the FPGA chip may not have any more resources on-board and thus will fail in creating the bitstream in the next stage. Unrolling the inner most loops is always done.

Wherever possible, unrolling was preferred over pipelining as performances are much better.

5.4.4 Accelerator Specifications

There are in total four accelerators which were developed. First of which is the GRUCell, the program code can be found in Appendix L. Second is the weight normalisation for *v* of size 64x64x1, code shown in Appendix M. The third accelerator is once again, weight normalisation but with *v* of size 256x64x1, code shown in Appendix N. Finally, an accelerator which handles a combination of both the stated weight normalisation dimensions, the program for which is still the same as the previous ones but with slight modifications to the unrolling factors to allow the designs to fit together.

As for the accelerator which consists of the combination of two dimensions for weight normalisation, priority was given to the larger dimension to use more space as the CPU struggles to perform matrix multiplication for larger sizes than smaller ones. In addition, the Vivado 2018.2 design for this accelerator varies slightly as there exists two DMAs. This has been shown in Appendix O. Each DMA is assigned an equal share of the total 4GB from the Address Editor, used previously. As for the Vivado 2018.2 designs for the other accelerators, no changes were made from the generic design discussed earlier (Appendix D).

When it comes to embedding the accelerators within SampleRNN, the exact same concept was used as shown in the RNN layer example. No accelerator was used together as the time to download a new bitstream on the FPGA chip takes exactly 400ms according to the work done by Hao and Quigley (2017).

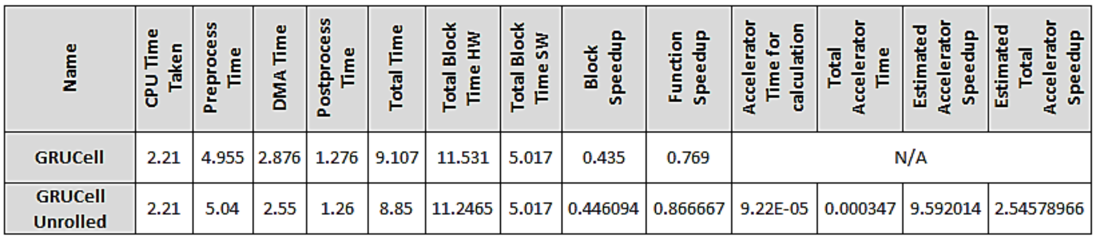
6 RESULTS & ANALYSIS

Within this chapter, the performance of the accelerators embedded within SampleRNN are presented and analysed.

At first, line profiling was conducted for each individual accelerator, results for which have been tabulated. Since various versions of the accelerator were investigated on, the tables showcase these in order of development but the fastest among the lot being further analysed. The metrics shown are defined as follows: The CPU Time Taken refers to the amount of time taken by the CPU to compute the exact processing as what the accelerator was designed to do. The Pre-process Time is the amount of time spent in making the data appropriate to be passed onto the DMA. DMA Time accounts for the transferring and waiting times while using the DMA API. The Post-processing Time refers to time spent within commands to bring back data into a form which can be used by the program again. Total Time is the addition of the pre-processing, DMA, and post-processing times. Total Block Time HW refers to the total time spent in the kernel after adding the hardware whereas, Total Block Time SW is GRUCell function when using only the CPU. Block Speedup is kernel on CPU divided by kernel on hardware execution times. Function Speedup is the ratio between CPU Time taken and DMA Time. Next, the Accelerator Time for calculation column specifies the amount of time estimated by Vivado HLS simulation, which only for the processing of a single call of the accelerator (without stream I/O). Note that during execution, there are multiple calls to the accelerator. The Total Accelerator Time is another simulation result by Vivado HLS but for the complete design. The last two columns are ratios between the CPU Time Taken and the values simulated by Vivado HLS for the processing only and the complete program, respectively. Since the values in Vivado HLS are for a single execution, the CPU Time Taken was divided by the number of times the function was called.

Table 4 contains the results for GRUCell. Two major versions of this accelerator were developed. As seen in the table, the speedup attained is 0.45x. This of course implies that the kernel has ultimately slowed down. However, the simulation results provided by Vivado HLS indicate that the processing was conducted almost 9.6x faster than the CPU, with an overall speedup of 2.55x. To identify what has caused the hardware to slowdown, a pie chart is shown in Figure 27. It seems that the majority amount of time is spent in pre-processing, which is already twice the time taken by the CPU to complete the execution. The mean-square error for this accelerator was 1.8602 x10^{-7}, providing high accuracy as the fixed-point datatype was set to FIX_32_16.

Table 4 – Performance metrics for GRUCell accelerators.



Name	CPU Time Taken	Preprocess Time	DMA Time	Postprocess Time	Total Time	Total Block Time HW	Total Block Time SW	Block Speedup	Function Speedup	Accelerator Time for calculation	Total Accelerator Time	Estimated Accelerator Speedup	Estimated Total Accelerator Speedup
GRUCell	2.21	4.955	2.876	1.276	9.107	11.531	5.017	0.435	0.769	N/A			
GRUCell Unrolled	2.21	5.04	2.55	1.26	8.85	11.2465	5.017	0.446094	0.866667	9.22E-05	0.000347	9.592014	2.54578966

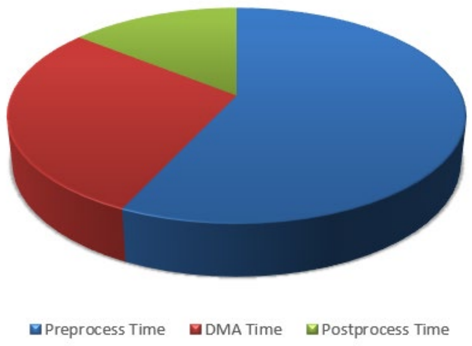
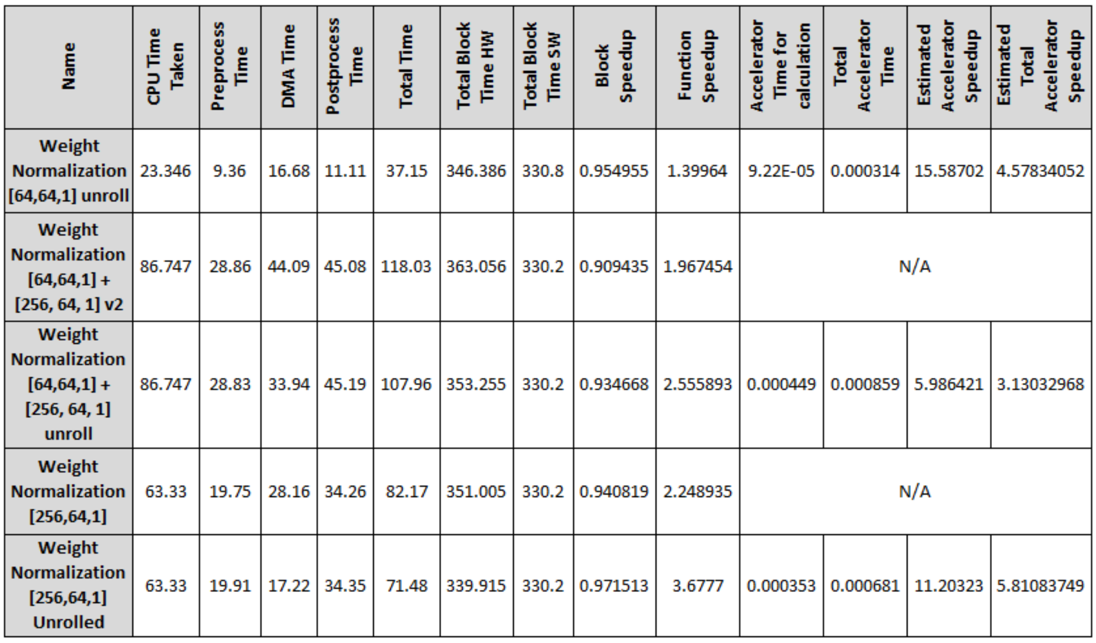


Figure 27 - GRUCell timing breakdown.

Moving on, the results of the three accelerators for weight normalisation have been shown in Table 5. Once again, different design improvements were made for the same accelerator, this involved improving unrolling factors and software optimizations. The speedup factor for weight normalisation for 64x64x1, 256x64x1, and the combination of the two are 0.91x, 0.97x, and 0.94x, respectively. On the other hand, the simulation speedup for these were 15.59x, 11.20x, and 5.99x, respectively. As for the accuracy of the accelerators, the mean-square error is 0.1974 for all. Once again, the accelerator has slowed down the execution speed. The timing breakdown of the three accelerators is shown in Figure 28. The breakdown clearly shows that the accelerator is being overshadowed by the pre/post-processing once again.

Table 5 - Performance metrics for Weight Normalization accelerators.



Name	CPU Time Taken	Preprocess Time	DMA Time	Postprocess Time	Total Time	Total Block Time HW	Total Block Time SW	Block Speedup	Function Speedup	Accelerator Time for calculation	Total Accelerator Time	Estimated Accelerator Speedup	Estimated Total Accelerator Speedup
Weight Normalization [64,64,1] unroll	23.346	9.36	16.68	11.11	37.15	346.386	330.8	0.954955	1.39964	9.22E-05	0.000314	15.58702	4.57834052
Weight Normalization [64,64,1] + [256, 64, 1] v2	86.747	28.86	44.09	45.08	118.03	363.056	330.2	0.909435	1.967454	N/A			
Weight Normalization [64,64,1] + [256, 64, 1] unroll	86.747	28.83	33.94	45.19	107.96	353.255	330.2	0.934668	2.555893	0.000449	0.000859	5.986421	3.13032968
Weight Normalization [256,64,1]	63.33	19.75	28.16	34.26	82.17	351.005	330.2	0.940819	2.248935	N/A			
Weight Normalization [256,64,1] Unrolled	63.33	19.91	17.22	34.35	71.48	339.915	330.2	0.971513	3.6777	0.000353	0.000681	11.20323	5.81083749

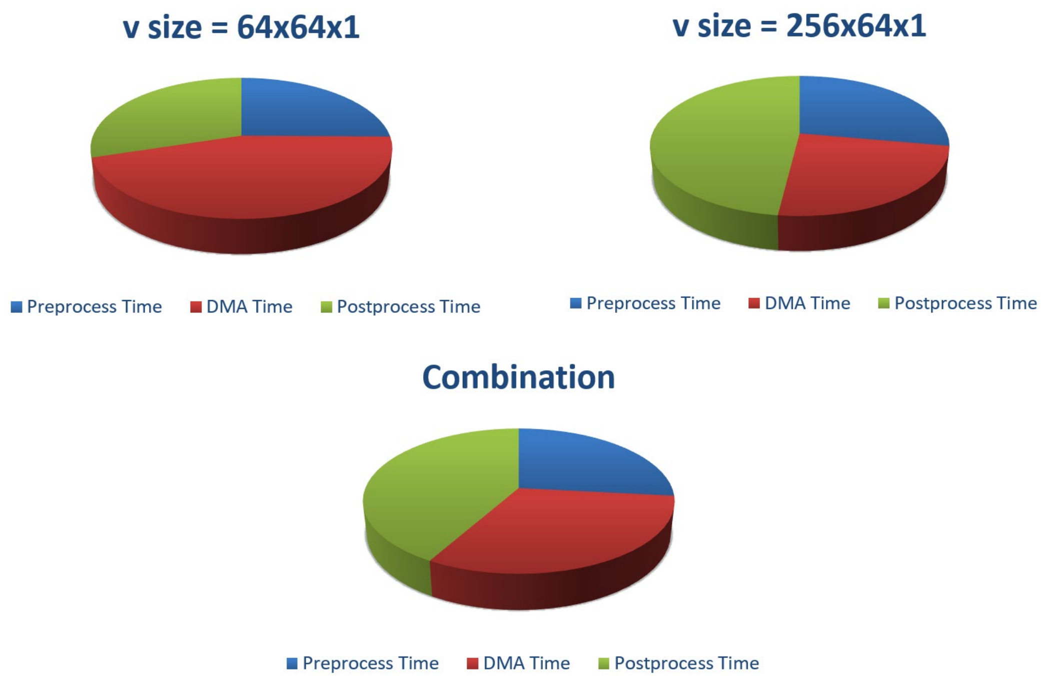
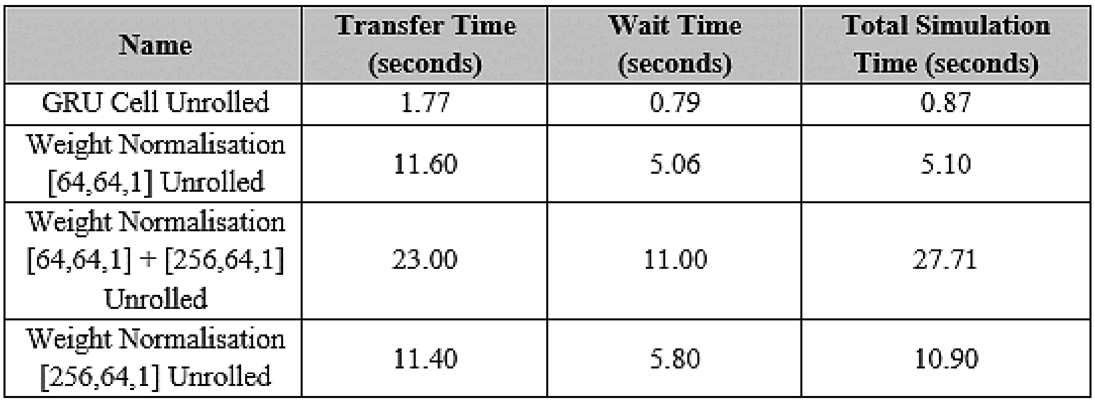


Figure 28 - Weight Normalization timing breakdown.

Moreover, in all accelerators designed, the simulation results do not replicate when embedded within the python program. When embedding the accelerators, a new bottleneck in the system is observed. This includes the pre/post-processing to format the data, as well as the DMA. On average, with respect to the total time spent when using the accelerator, the pre-processing accounts for 36.1%, whereas post-processing amounts to 28.8%. Further understanding of the DMA API may be important as the transfer and wait functions do not matching the simulation results. The transfer simply forwards the physical address of the contiguous array created at first in pre-processing, yet it consumes a significant amount of execution time. The wait function stops the program from proceeding until the data is received from the accelerator. It is unclear whether the hardware begins as soon as transfer function is called or when wait has started. To further investigate the above issue, Table 6 provides a comparison of the simulation results and time spent within the transfer and wait functions. Clearly, when utilizing the DMA API, hidden latencies are seen, which must be discovered and improved as future works, in order to further enhance the framework.

Table 6 - Comparison of DMA API with simulation results.



Name	Transfer Time (seconds)	Wait Time (seconds)	Total Simulation Time (seconds)
GRU Cell Unrolled	1.77	0.79	0.87
Weight Normalisation [64,64,1] Unrolled	11.60	5.06	5.10
Weight Normalisation [64,64,1] + [256,64,1] Unrolled	23.00	11.00	27.71
Weight Normalisation [256,64,1] Unrolled	11.40	5.80	10.90

7 CONCLUSIONS

Sound synthesis is a technique whereby analog or digital circuits are used to generate sound. There are many different types of synthesizers, such as the simple traditional ones where periodic waveforms are generated at different frequencies, and more complicated ones, such as physical modelling, where mathematical models are developed to mimic instruments and other acoustic bodies. Research has led to efficient algorithms which mainly aim to reduce the number of high-quality waveforms produced or completely nullify waveforms due to lack of difference in perceived audio.

NNs are computational structures which contain neurons organized within layers. This forms a mesh network with weighted connections, a bias inside each neuron along with a non-linear thresholding function. When a NN is trained with audio data, it is able to produce related sound in a unique pattern. This has led to NNs being utilized as sound synthesizers. However, any NN typically requires heaps of calculations to be processed before producing a single output, which poses as a challenge within sound synthesis. To produce high-quality sound, at least 16,000 samples per second are needed, where state-of-the-art models have provided barely 450 samples per second on a high-end PC. This implies the need for an accelerator to speed up the kernel.

Literature has shown that FPGA based accelerators for NNs have exemplary performances. These are reconfigurable hardware which perform any function a developer designs it for, providing parallel computational capabilities. Although not a common choice among developers due to expert design knowledge required, this work presents a framework which uses the Xilinx PYNQ-Z1 board to ease the development process. A standard RNN layer was accelerated by 21 times using the framework, showing promising results. On the other hand, an RNN based sound synthesizer, SampleRNN, is also accelerated. The program was profiled at first, where bottle-necks were identified to be two kernels related to GRUs and weight normalisation. These were redesigned using hardware on the FPGA, aiming to accelerate the computation. The design is one-of-a-kind hybrid model which uses both CPU and FPGA. SampleRNN was not accelerated as per the goal set initially which was about 10 times however, simulation results did illustrate the performance of the hardware providing speedup factors from 9.96x to 15.59x. As for future works, a far more efficient method to transfer data to the hardware must be investigated upon. Since when embedding the accelerator, instead of eradicating the bottlenecks, new ones were created in the form of pre/post-processing and the DMA API. On average, pre-processing accounts for 36.1% of accelerator execution time whereas the post-processing occupies 28.8%. Furthermore, at times, the PYNQ-Z1 board FPGA fell short on space, which made the hardware compensate, hence a larger FPGA size may also be investigated on PYNQ boards. Lastly, a kernel not investigated in SampleRNN were the convolutional layers within the lower tiers as it was out of the scope of this research but indeed it still remains as a bottleneck to be investigated as future works.

8 REFERENCES

Chang, A.X.M., Martini, B., Culurciello, E., 2015. Recurrent Neural Networks Hardware Implementation on FPGA. ArXiv151105552 Cs.

Chang, S., Zhang, Y., Han, W., Yu, M., Guo, X., Tan, W., Cui, X., Witbrock, M., Hasegawa-Johnson, M., Huang, T.S., 2017. Dilated Recurrent Neural Networks. ArXiv171002224 Cs.

Chung, J., Gulcehre, C., Cho, K., Bengio, Y., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. ArXiv14123555 Cs.

Digilent Inc, 2018. PYNQ-Z1 Reference Manual [Reference.Digilentinc] [WWW Document]. URL https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual (accessed 9.29.18).

Finnerty, A., Ratigner, H., 2017. Reduce Power and Cost by Converting from Floating Point to Fixed Point.

Graves, A., 2013. Generating Sequences With Recurrent Neural Networks. ArXiv13080850 Cs.

Guan, Y., Yuan, Z., Sun, G., Cong, J., 2017. FPGA-based accelerator for long short-term memory recurrent neural networks, in: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). Presented at the 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, Chiba, Japan, pp. 629–634. https://doi.org/10.1109/ASPDAC.2017.7858394

Hao, Y., Quigley, S., 2017. The implementation of a Deep Recurrent Neural Network Language Model on a Xilinx FPGA. ArXiv171010296 Cs.

Hind, N., 2018. Physical Modelling Synthesis [WWW Document]. URL https://ccrma.stanford.edu/software/clm/compmus/clm-tutorials/pm.html (accessed 11.17.18).

Huynh, T.V., 2017. Deep neural network accelerator based on FPGA, in: 2017 4th NAFOSTED Conference on Information and Computer Science. Presented at the 2017 4th NAFOSTED Conference on Information and Computer Science, pp. 254–257. https://doi.org/10.1109/NAFOSTED.2017.8108073

IEEE, 2008. 754-2008 - IEEE Standard for Floating-Point Arithmetic - IEEE Standard [WWW Document]. URL https://ieeexplore-ieee-org.ezproxy.uow.edu.au/document/4610935 (accessed 3.29.19).

Imperial College London, 2018. Fourier Series [WWW Document]. Fourier Ser. URL http://wwwf.imperial.ac.uk/metric/metric_public/fourier_theory/series/fourier_series.html (accessed 11.17.18).

Intel®, 2018. Intel® FPGAs Powering Real-Time AI Inferencing [WWW Document]. Intel AI. URL https://ai.intel.com/intel-fpgas-powering-real-time-ai-inferencing/ (accessed 9.29.18).

ITU, 2016. P.800.1 : Mean opinion score (MOS) terminology [WWW Document]. URL https://www.itu.int/rec/T-REC-P.800.1 (accessed 12.7.18).

Johnson, J., 2017. Lecture 10 | Recurrent Neural Networks.

Kalchbrenner, N., Elsen, E., Simonyan, K., Noury, S., Casagrande, N., Lockhart, E., Stimberg, F., Oord, A. van den, Dieleman, S., Kavukcuoglu, K., 2018. Efficient Neural Audio Synthesis, in: ArXiv:1802.08435 [Cs, Eess].

Kern, R., 2019. Line-by-line profiling for Python. Contribute to rkern/line_profiler development by creating an account on GitHub.

Kozakowski, P., Kańska, K., Rishaug, J., 2017. PyTorch implementation of SampleRNN: An Unconditional End-to-End Neural Audio Generation Model: deepsound-project/samplernn-pytorch. DeepSound.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., Quillen, D., 2018. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. Int. J. Robot. Res. 37, 421–436.

Li, F.-F., 2017. Lecture 4 | Introduction to Neural Networks.

LibROSA, 2018. LibROSA — librosa 0.6.2 documentation [WWW Document]. URL https://librosa.github.io/librosa/ (accessed 12.4.18).

McCulloch, W.S., Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. 5, 115–133. https://doi.org/10.1007/BF02478259

Mehri, S., Kumar, K., Gulrajani, I., Kumar, R., Jain, S., Sotelo, J., Courville, A., Bengio, Y., 2016. SampleRNN: An Unconditional End-to-End Neural Audio Generation Model.

Mohammadi, M., Mundra, R., Socher, R., n.d. CS 224D: Deep Learning for NLP.

Narang, S., Elsen, E., Diamos, G., Sengupta, S., 2017. Exploring Sparsity in Recurrent Neural Networks, in: ArXiv:1704.05119 [Cs].

Nave, R., 2018. Sensitivity of Human Ear [WWW Document]. URL http://hyperphysics.phy-astr.gsu.edu/hbase/Sound/earsens.html#c2 (accessed 12.7.18).

NumPy, 2019. NumPy — NumPy [WWW Document]. URL http://www.numpy.org/ (accessed 4.2.19).

Oh, K.-S., Jung, K., 2004. GPU implementation of neural networks. Pattern Recognit. 37, 1311–1314. https://doi.org/10.1016/j.patcog.2004.01.013

Oktay, O., Ferrante, E., Kamnitsas, K., Heinrich, M., Bai, W., Caballero, J., Cook, S.A., de Marvao, A., Dawes, T., O'Regan, D.P., others, 2018. Anatomically constrained neural networks (ACNNs): application to cardiac image enhancement and segmentation. IEEE Trans. Med. Imaging 37, 384–395.

Oord, A. van den, Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., Kavukcuoglu, K., 2016. WaveNet: A Generative Model for Raw Audio. ArXiv160903499 Cs.

Paine, T.L., Khorrami, P., Chang, S., Zhang, Y., Ramachandran, P., Hasegawa-Johnson, M.A., Huang, T.S., 2016. Fast Wavenet Generation Algorithm.

PYNQ, 2018. PYNQ - Python productivity for Zynq [WWW Document]. PYNQ - Python Product. Zynq. URL http://www.pynq.io/home.html (accessed 12.7.18).

Python Documentation, 2018. The Python Profilers — Python 3.7.1 documentation [WWW Document]. URL https://docs.python.org/3/library/profile.html#module-profile (accessed 11.16.18).

PyTorch, 2018a. PyTorch [WWW Document]. URL https://www.pytorch.org (accessed 9.30.18).

PyTorch, 2018b. torch.nn — PyTorch master documentation [WWW Document]. URL https://pytorch.org/docs/stable/nn.html#weight-norm (accessed 12.4.18).

PyTorch Documentation, 2019. torch.nn — PyTorch master documentation [WWW Document]. URL https://pytorch.org/docs/0.4.1/nn.html#rnn (accessed 4.2.19).

PyTorch GRUCell Documentation, 2019. torch.nn — PyTorch master documentation [WWW Document]. URL https://pytorch.org/docs/0.4.1/nn.html#grucell (accessed 4.5.19).

Raghuvanshi, N., Lauterbach, C., Chandak, A., Manocha, D., Lin, M.C., Lin, M.C., 2007. Real-time Sound Synthesis and Propagation for Games. Commun ACM 50, 66–73.

Rise, S., 2014. Wavetable Synthesis | The Synthesizer Academy [WWW Document]. URL http://synthesizeracademy.com/wavetable-synthesis/ (accessed 11.17.18).

Schwarz, D., 2007. Corpus-Based Concatenative Synthesis. IEEE Signal Process. Mag. 24, 92–104. https://doi.org/10.1109/MSP.2007.323274

Serra, X., 2007. State of the Art and Future Directions in Musical Sound Synthesis, in: 2007 IEEE 9th Workshop on Multimedia Signal Processing. Presented at the 2007 IEEE 9th Workshop on Multimedia Signal Processing, pp. 9–12. https://doi.org/10.1109/MMSP.2007.4412805

Smith III, J.O., 2018a. Additive Synthesis (Early Sinusoidal Modeling) [WWW Document]. URL https://ccrma.stanford.edu/~jos/sasp/Additive_Synthesis_Early_Sinusoidal.html (accessed 12.7.18).

Smith III, J.O., 2018b. Basics of Digital Waveguide Modeling [WWW Document]. URL https://ccrma.stanford.edu/~jos/swgt/Basics_Digital_Waveguide_Modeling.html (accessed 10.24.18).

Stanford University, 2018. Unsupervised Feature Learning and Deep Learning Tutorial [WWW Document]. URL http://deeplearning.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/ (accessed 12.7.18).

TensorFlow, 2018. TensorFlow [WWW Document]. TensorFlow. URL https://www.tensorflow.org/ (accessed 10.1.18).

Tian, Y., Pei, K., Jana, S., Ray, B., 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars, in: Proceedings of the 40th International Conference on Software Engineering. ACM, pp. 303–314.

Vivado, 2018. Vivado Design Suite [WWW Document]. URL https://www.xilinx.com/products/design-tools/vivado.html (accessed 9.29.18).

Wyse, L., 2018. Real-valued parametric conditioning of an RNN for interactive sound synthesis.

Xilinx, 2019. HLS Pragmas [WWW Document]. URL https://japan.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html (accessed 3.30.19).

Xilinx, 2018a. What is an FPGA? Field Programmable Gate Array [WWW Document]. URL https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html (accessed 12.8.18).

Xilinx, 2018b. Vivado Design Suite User Guide: High-Level Synthesis (UG902).

Xilinx, 2011. AXI Reference Guide 13, 82.

APPENDIX A REVISED PROJECT PROPOSAL

1. Candidate Details	
Name: Manohar Vohra	**Student No: 5265071**
Supervisor: Dr. Stefano Fasciani	
Title of Project: Hardware Accelerator for Recurrent Neural Network-based Sound Synthesis	
Brief Overview: Recently, Google has introduced WaveNet, which uses Neural Networks (NNs) for raw audio signal generation [1], [2]. The combination of NNs with sound synthesis implies that machines are trained with millions of audio examples allowing them to learn and synthesize sound as speech or music at single sample level. Although optimization efforts have been considered through different algorithms and hardware previously (i.e. Graphics Processing Units) [3]–[10], none are capable of performing synthesis in real-time. Using hardware accelerators to speed up a NNs computation is an upward trend in recent years [8]. This project aims to design a hardware accelerator to speed up the computation of a recurrent neural network (RNN) used in sound synthesis. In particular, this project will introduce a framework profiling the computational load of existing RNN applications and replace the critical functions with dedicated hardware synthesized on a Field Programmable Gate Array (FPGA). While RNNs are well-known for sequential-based learning [11]–[14], the challenge here would be that such a network cannot generate audio at the pace required since the samples are generated one at time. Previously, other applications have used FPGAs to accelerate NNs [15]–[17], providing promising results, which is aimed to be replicated within this application context.	

2. Project Description:

Sound synthesis is a technique to generate raw audio waveforms using computer hardware/and or software [18] seen in many applications such as text-to-speech (TTS), video games, films, music, electronic musical instruments, etc [19]–[22]. Most of these real-world applications require sound synthesizers to operate under real-time constraints, which indeed includes being fast. A typical example here is Granular Synthesis, which was only adopted once it was real-time [23]. Moreover, for consumer applications it is also important to be cost-effective and maintaining reasonable power consumption [24].

A Neural Network (NN) is an architecture containing layers of neurons, namely input, hidden, and output layers. These are interconnected together to form a mesh network, where each connection has a weight associated with it. The weights are multiplied with the input to this connection and given to the next neuron. Neurons contain non-linear activation functions with thresholds which indicate whether or not a certain characteristic has been identified from the input [25]–[27]. The threshold (bias) and weights are adjusted during training periods, to be able to give the maximum accuracy possible to the output. The aim of training the network is to present a large set of inputs along with the desired output, and allow the machine to be able to estimate a model embedded in the training data [28], [29].

The problem being addressed within this project remains unchanged from ECTE451, where there exists a lack of research investigating how recurrent-neural-network-based sound synthesizers can be implemented for real-time applications. Through work already completed thus far, it is only further deduced that such systems are yet to be real-time. Within the implementations seen in [2], [4], [30] and others, credible results have been achieved in terms of functionality. But, in order to be real-time oriented, these systems need to take into account the speed of generating audio, whilst operating within the limited resource constraints. If such systems with these capabilities were to be implemented using computers of today, they would require an accelerator on an Application Specific Integrated Chip (ASIC) alongside the CPU, which is not feasible. Instead, the accelerator can be designed and implemented on an FPGA, which provide flexibility and high computational power [31]. Although FPGA design is challenging, the proposed framework integrates an existing deep learning programming environment with FPGA deployment tools. This simplifies designing and deploying accelerators for NNs, allowing developers to implement and evaluate application-specific accelerators.

The framework will be based on the Xilinx PYNQ-Z1 board [32]. The Systems on Chip (SoC) uniquely provides a single platform solution containing both a microcontroller, and an FPGA. Integration between the programmable (microcontroller) and reconfigurable (FPGA) components is done through a concept of a software library labelled as Overlays. In conjunction with a powerful Python developing environment and Debian-like Operating System, the board places itself perfectly for the purpose. The board has been used to accelerate Deep Recurrent Neural Networks (DRNN is a NN with multiple hidden layers) in literature [31]. What the authors of this paper have failed to take into account is one of the most competitive NN libraries, PyTorch [33]. This thesis will also be the first to execute PyTorch-written code on the PYNQ-Z1, which is expected to open a gateway of new applications and their enhancements proving the significance of this work.

Below are the objectives and outcomes of this work. Note that the points in italic are to be completed within the session, whereas the remaining have already been completed in the previous session.

Objectives of this project:

1. To interpret an existing RNN-based sound synthesizer by top-level profiling, leading to the identification of computationally demanding kernels. (Completed)
    - Illustrate how slow current RNN-based sound synthesizers are. (Completed)
    - This objective is extended to also include the line profiling of those kernels flagged by the top-level profiling to be system bottlenecks. The need for such in-depth profiling had been discovered through the research work conducted thus far, as doing so will allow further understanding as to what should be modelled in hardware later. (Completed)
2. *Compose a computationally efficient architecture for the RNN kernel.*
3. *Develop a framework to evaluate and test the RNN-based synthesis acceleration. Notably:*
    - *Port (compile and execute) an existing deep learning environment on an FPGA-based platform.*
    - *Model the designed architecture on an FPGA.*
    - *Create a new program flow by integrating the CPU and FPGA computations together.*

Outcomes of this project:

1. Profile an existing RNN based sound synthesis and identify the modules presenting high computational cost. (Completed)
2. *Design and implement such modules as an FPGA-based accelerator integrated in a computing system.*
3. *Assess the performance of the RNN based sound synthesis running with and without accelerator.*
4. *Develop a framework which enhances performance of RNN-based algorithms by integrating custom accelerators for FPGA in an existing deep learning programming environment.*

3. Project Plan:

Within the scope of this project, a hardware accelerator will be designed on an FPGA for sound synthesizers which utilize RNNs. The aim here would be to improve the speed and resource usage after performing detailed analysis. Since projects may encounter various dead ends, listed below are methods (preference-wise) which are devised as contingency plans. Based on the work completed in the previous session, Method 1, which was the most preferred method, will be the plan which will be followed. The reasoning behind this selection is due to the successful installation of PyTorch on the chosen platform. Again, the work to be completed within this session are in italic.

- Method 1
    - PyTorch must be successfully installed. (Completed)
    - Compile a sound synthesizer which uses RNNs, for instance SampleRNN [11]. (Completed)
    - Test the speed, power and accuracy currently achieved on the board. (Completed)
    - Identify what needs to be accelerated. This will be conducted by two layers of profiling. (Completed)
        - Top-level: profiling the code using software tools with details only provided at function level. (Completed)
        - In-depth: once the demanding functions are identified, they will be profiled at line-level. (Completed)
    - *Analyse whether that operation/s can be designed on an FPGA. Must answer the following:*
        - *Where are inputs coming from? This will help in identifying on which data will the operations be performed on as well as their datatype.*
        - *What processing is to be conducted? Involves studying what all do the above inputs undergo to attain the outputs.*
        - *Where should the outputs be placed? Once the hardware processes the input data, where must this be placed to continue the program flow.*
        - *Feasibility of the kernel will depend on these three factors above.*
    - *If feasible, proceed with designing. Else, look for another operation which also consumes time, and attempt to accelerate that.*
    - *Test the new system.*
    - *Compare if a difference in performance is positive or not. If it is not positive, revert to design phase.*
    - *Create a general framework for designing the accelerator. This will include what operations can be accelerated, the workflow involved, and generic designs proposed.*
    - *Finally, make the findings available to the community.*
    - *As for the developed hardware/software in this method, the hardware is soft-core, meaning inside the FPGA, plus, there will also be a C++ program which will be converted to hardware using specialised software tools. Further, code alterations may also be done for the selected existing application so as to meet the budget constraints (i.e. if the model is very large to design on the FPGA, the size will be reduced).*
- Method 2 (No longer considered)
    - If PyTorch does not successfully compile on the PYNQ-Z1 board, another famous NN library will be utilised, known as TensorFlow [34]. PyTorch is simpler to use and easy for researchers to work with although TensorFlow is competitive as well and has been compiled on the PYNQ-Z1 previously [35], [36].
    - The same steps will then be followed as before but applications written with TensorFlow will be selected.

- Method 3 (No longer considered)
    - Another option is to continue using PyTorch, but instead of running it on the PYNQ, it will be executed on another board, preferably the Raspberry Pi 3 B+.
    - Now then, there are a few differences in the architecture of the framework proposed within this method. The application will be running on the Pi instead of the PYNQ, however, after finding out the bottle-necks within the application, the FPGA will be designed on the PYNQ, and utilizing communication through the internet, the inputs/outputs can be shared with the Pi.
    - Hence, the execution will be as such, the Pi will run the program. When it reaches the segment found to be slow, it will send the inputs required to the PYNQ and the FPGA will be processing the inputs. Finally, the outputs will be sent back to the Pi and it will continue to execute the code.
    - During analysis of performance, the communication latency will be disregarded.
    - In addition, a new software will be programmed to handle the communication mappings.

The strategy of using FPGAs has been exploited here due to their promising results in previous findings. For instance, in [16], the author uses an FPGA to accelerate a Convolutional Neural Network (CNN). The design here somewhat resembles the system proposed within this project, though, instead of obtaining data from a memory chip, the data in this research is going to fed by the CPU. A gap in this paper would be that it lacks to maximize the parallelism within process two and three, as it performs these in a sequential manner. Even though utilization rates were not quite as high, performances were far better than CPU and GPU execution, which proves the effectiveness of FPGAs.

In contrast to the previous paper, [15] has accelerated a DNN. As opposed to a CNN, a DNN requires far more computation, which might explain the difference in utilization rates. Note that the system architecture presented here has no CPU once again. Moreover, the paper proposes a forward propagation of the network in a sequential manner, which, again, can affect the speed, and is found to be a gap. It is worth mentioning that the design proposed here is compact and simple, although better performances have been achieved.

Another key paper [31] proposes an accelerator on the PYNQ-Z1 board. The paper has provided a methodology with components proposed here, attaining up to twice the speed of a GPU. A point to consider here is that the model creates the complete network on the FPGA but fails to utilize the CPU on-board (only used for training).

The model proposed for this research project will run as a hybrid, exploiting both the CPU and the FPGA. What the above stated papers lack is the execution time as comparison, which will be a key attribute when comparing models to understand real-time compatibility. Hence, results will be measured by recording how long it has taken to execute the program without the accelerator and then with the accelerator. In order to validate the implementation, we compare data processed by the accelerator against the data processed by the CPU. As the accelerator is implemented in fixed point architecture, we need to ensure that there is sufficient accuracy in the generated result. As an additional result, comparisons with GPU and CPU execution will also be considered.

On the other hand, it is noteworthy to say that the proposed framework will not be speeding up the training period. Training periods are usually time-consuming however, this can be done before deployment of the application and so will not have an effect on the run-time performance (although accuracy depends on how well trained the net is).

Thus far, the status of the project is as follows:

- ✓ Initially, the Regression example program provided on the GitHub page of PyTorch [37] was studied and accelerated. This was done in order to further consolidate the methodology which will be used when accelerating the chosen synthesizer application. In the process of doing so, it was discovered that the top-level profiling was not enough to understand the bottlenecks of the system. In addition, the approach to design the hardware will include using Vivado HLS [38] for writing C++ programs, which is then synthesized. This is then exported to as IP and connected within a model on Vivado 2018.2 [39]. This application was accelerated such that on average, the training period will execute 2.35x faster. This design here calculated the following equations, which is the gradients followed by matrix multiplications (not shown here):

$$\boldsymbol{gradient} = \begin{cases} \frac{\mathbf{1}}{\boldsymbol{n}}(\boldsymbol{x}-\boldsymbol{y}), & |\boldsymbol{x}-\boldsymbol{y}| < \mathbf{1} \\ \frac{\mathbf{1}}{\boldsymbol{n}}(\frac{\boldsymbol{x}-\boldsymbol{y}}{|\boldsymbol{x}-\boldsymbol{y}|}), & \boldsymbol{otherwise} \end{cases} \qquad \textbf{(1)}$$

- ✓ Through the above experiment, methods to transfer larger data efficiently was also discovered. The two methods are either using the, AXI4Lite interface [40] or using AXI-Stream interface [40]. Using the AXI-Stream utilizes the Direct Memory Access (DMA) IP and is the best approach when large amount of data is to be transferred.
- ✓ Next, since the previous application was accelerating the training period, another application was programmed such that the accelerator will closely resemble what the synthesizer accelerator will include. This was a simple program developed which included a two-layer RNN. The network did not have any specific functionality but instead, the aim was to imagine that once an RNN was trained, can an accelerator be designed to compute the output when provided with a random input. Hence, rather than the functionality of the network being something specific, the focus was on the operations involved. This included the follow equation taken from the PyTorch documentation [41]:

$$\boldsymbol{h_t} = \mathbf{tanh}(\boldsymbol{w_{ih}x_t} + \boldsymbol{b_{ih}} + \boldsymbol{w_{hh}h_{(t-1)}} + \boldsymbol{b_{hh}}) \qquad \textbf{(2)}$$

  The application with the accelerator had a speedup factor of almost 13x. The dimension, which is the size of the inputs, was 128. The smaller this is, the greater the acceleration attained. Another outcome of this experiment was that the overhead of transferring large sums of data is positively correlated to the time taken to transfer. This indeed becomes challenge which must be faced later in the project. In addition, if a larger FPGA chip was to be used, a significantly higher speedup factor could have been achieved as more parallel units could have been deployed although this is a budget constraint to the project.
- ✓ A side test to the above was conducted regarding the speed at which hardware performs the hyperbolic tangent calculation. Results indicated it was 20% slower than the CPU.
- ✓ As for the chosen application, SampleRNN, models have been training in the background on a server. Many variations, such as a new instrument, the guitar, has been put to train as well. Another important variation is the reduction of the dimension size mentioned in the previous experimental application as

well. This was done following a test which indicated that the application was using the SWAP at the default dimension size. Therefore, this must be avoided as it can be another bottleneck within the system. The default dimension being used was 1024, whereas new models being trained are for 256 and 512. Results for the 256 indicate no SWAP usage. As for 512, training is yet to reach an epoch.

A Gantt chart (Figure 1) indicating the completion of project objectives and tasks during ECTE458 can be found in the Appendix of this document.

Additional Materials from the ECTE451 Proposal are tabulated below:

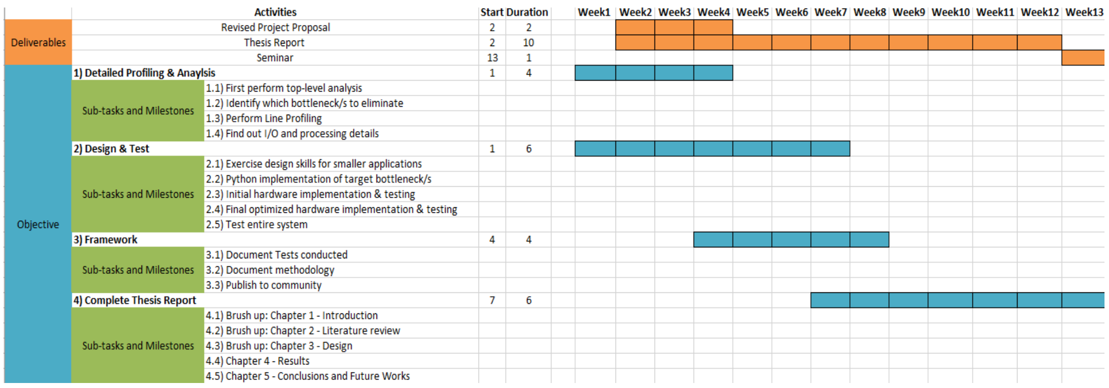
Table 1 - Additional materials required.

Resource Name/Description	Use-case within this project
Linux Server	For continuously training different neural networks in the background. When sound is to be generated, the epoch data (weights and bias) are exported on to the PYNQ-Z1 board.
PyTorch [42]	Neural Network library for Python. This is basis upon which the chosen application was built on. It is an open-source library which was ported on various machines within this project.
SampleRNN [43]	The chosen RNN-based sound synthesizer which will be at the core of this project.

4. Adaption of Supervisor and Examiners feedback in the ECTE451 report:

The comments provided in the previous session were mainly positive. Further details on some of the parameters of the Neural Network, such as the training loss, and their specific definitions within the context of this application will be provided in the final report this session. Lastly, the particulars on the architecture of SampleRNN will also be polished upon, as more attention will be paid within this session on the functionality while performing analysis.

Appendix



		Activities	Start	Duration	Week1	Week2	Week3	Week4	Week5	Week6	Week7	Week8	Week9	Week10	Week11	Week12	Week13
Deliverables		Revised Project Proposal	2	2		🟧	🟧	🟧									
		Thesis Report	2	10		🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	
		Seminar	13	1													🟧
Objective	1) Detailed Profiling & Anaylsis		1	4	🟦	🟦	🟦	🟦									
	Sub-tasks and Milestones	1.1) First perform top-level analysis															
		1.2) Identify which bottleneck/s to eliminate															
		1.3) Perform Line Profiling															
		1.4) Find out I/O and processing details															
	2) Design & Test		1	6	🟦	🟦	🟦	🟦	🟦	🟦	🟦						
	Sub-tasks and Milestones	2.1) Exercise design skills for smaller applications															
		2.2) Python implementation of target bottleneck/s															
		2.3) Initial hardware implementation & testing															
		2.4) Final optimized hardware implementation & testing															
		2.5) Test entire system															
	3) Framework		4	4				🟦	🟦	🟦	🟦	🟦					
	Sub-tasks and Milestones	3.1) Document Tests conducted															
		3.2) Document methodology															
		3.3) Publish to community															
	4) Complete Thesis Report		7	6							🟦	🟦	🟦	🟦	🟦	🟦	🟦
	Sub-tasks and Milestones	4.1) Brush up: Chapter 1 - Introduction															
		4.2) Brush up: Chapter 2 - Literature review															
		4.3) Brush up: Chapter 3 - Design															
		4.4) Chapter 4 - Results															
		4.5) Chapter 5 - Conclusions and Future Works															

Figure 1 - Gantt Chart indicating Milestones and Deliverables of ECTE458.

References

[1] "Google's DeepMind Claims Massive Progress in Synthesized Speech," *Fortune*. [Online]. Available: http://fortune.com/2016/09/09/google-deepmind-wavenet-ai/. [Accessed: 24-Sep-2018].

[2] A. van den Oord *et al.*, "WaveNet: A Generative Model for Raw Audio," *ArXiv160903499 Cs*, Sep. 2016.

[3] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[4] T. L. Paine *et al.*, "Fast Wavenet Generation Algorithm," Nov. 2016.

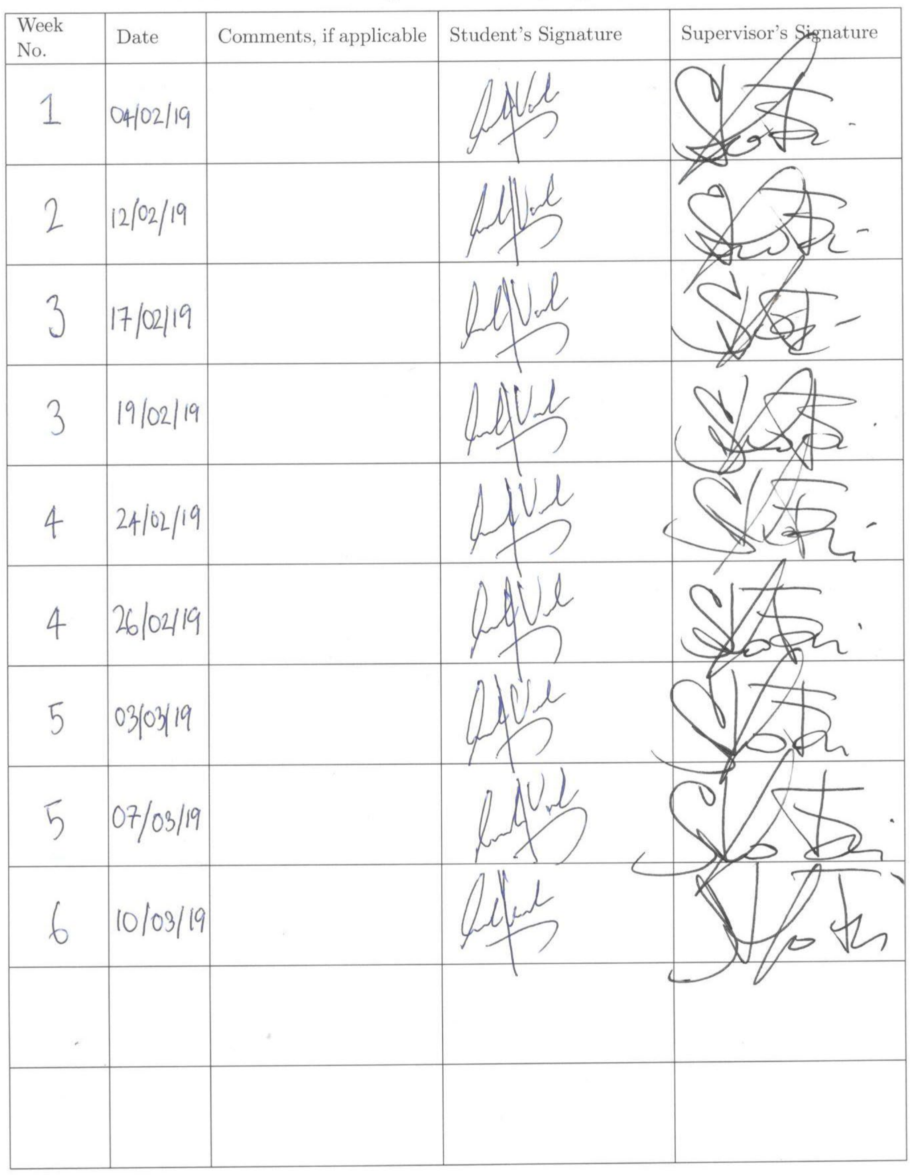
[5] D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling, "Improved Variational Inference with Inverse Autoregressive Flow," p. 9.

[6] D. Rausch, B. Hentschel, and T. Kuhlen, "Efficient modal sound synthesis on GPUs," in *2014 IEEE VR Workshop: Sonic Interaction in Virtual Environments (SIVE)*, 2014, pp. 13–18.

[7] N. Kalchbrenner *et al.*, "Efficient Neural Audio Synthesis," in *arXiv:1802.08435 [cs, eess]*, 2018.

[8] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognit.*, vol. 37, no. 6, pp. 1311–1314, Jun. 2004.
[9] L. Savioja, V. Välimäki, S. Iii, and J. O, "Real-Time Additive Synthesis with One Million Sinusoids Using a GPU," presented at the Audio Engineering Society Convention 128, 2010.
[10] "Realtime GPU Audio - ACM Queue." [Online]. Available: https://queue.acm.org/detail.cfm?id=2484010. [Accessed: 28-Sep-2018].
[11] S. Mehri *et al.*, "SampleRNN: An Unconditional End-to-End Neural Audio Generation Model," 2016.
[12] S. Chang *et al.*, "Dilated Recurrent Neural Networks," *ArXiv171002224 Cs*, Oct. 2017.
[13] A. Graves, "Generating Sequences With Recurrent Neural Networks," *ArXiv13080850 Cs*, Aug. 2013.
[14] "Recurrent neural network," *Wikipedia*. 14-Sep-2018.
[15] T. V. Huynh, "Deep neural network accelerator based on FPGA," in *2017 4th NAFOSTED Conference on Information and Computer Science*, 2017, pp. 254–257.
[16] Y. Huang, J. Shen, Z. Wang, M. Wen, and C. Zhang, "A High-efficiency FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm," in *Journal of Physics: Conference Series*, 2018, vol. 1026, p. 012019.
[17] Z. Liu *et al.*, "Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks," *ACM Trans Reconfigurable Technol Syst*, vol. 10, no. 3, pp. 17:1–17:23, Jul. 2017.
[18] "Sound Synthesis Theory/Introduction - Wikibooks, open books for an open world." [Online]. Available: https://en.wikibooks.org/wiki/Sound_Synthesis_Theory/Introduction. [Accessed: 28-Sep-2018].
[19] N. Raghuvanshi, C. Lauterbach, A. Chandak, D. Manocha, M. C. Lin, and M. C. Lin, "Real-time Sound Synthesis and Propagation for Games," *Commun. ACM*, vol. 50, no. 7, pp. 66–73, Jul-2007.
[20] S. Selvamani, "Development of a sound synthesis application and a music controller for sound designers," *RISE:2018*. .
[21] V. Välimäki, J. Huopaniemi, M. Karjalainen, and Z. Jánosy, "Physical Modeling of Plucked String Instruments with Application to Real-Time Sound Synthesis," presented at the Audio Engineering Society Convention 98, 1995.
[22] "Cloud Text-to-Speech - Speech Synthesis | Cloud Text-to-Speech API," *Google Cloud*. [Online]. Available: https://cloud.google.com/text-to-speech/. [Accessed: 29-Sep-2018].
[23] B. Truax, "Real-Time Granular Synthesis with a Digital Signal Processor," *Comput. Music J.*, vol. 12, no. 2, pp. 14–26, 1988.
[24] J. A. Stankovic, "Real-time and embedded systems," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 205–208, Mar. 1996.
[25] S. Steinke, "What's the difference between a matrix and a tensor?," *Medium*, 28-Aug-2017. .
[26] A. S. V, "Understanding Activation Functions in Neural Networks," *Medium*, 30-Mar-2017. .
[27] "Autograd: Automatic Differentiation — PyTorch Tutorials 1.0.0.dev20181002 documentation." [Online]. Available: https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html. [Accessed: 04-Oct-2018].
[28] F.-F. Li, "Lecture 4 | Introduction to Neural Networks," Stanford University School of Engineering, 11-Aug-2017.
[29] "Artificial neural network," *Wikipedia*. 28-Sep-2018.
[30] L. Wyse, "Real-valued parametric conditioning of an RNN for interactive sound synthesis," 2018.
[31] Y. Hao and S. Quigley, "The implementation of a Deep Recurrent Neural Network Language Model on a Xilinx FPGA," *ArXiv171010296 Cs*, Oct. 2017.
[32] Digilent Inc, "PYNQ-Z1 Reference Manual [Reference.Digilentinc]," 2018. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/reference-manual. [Accessed: 29-Sep-2018].
[33] PyTorch, "PyTorch," 2018. [Online]. Available: https://www.pytorch.org. [Accessed: 30-Sep-2018].
[34] TensorFlow, "TensorFlow," *TensorFlow*, 2018. [Online]. Available: https://www.tensorflow.org/. [Accessed: 01-Oct-2018].
[35] K. Dubovikov, "PyTorch vs TensorFlow — spotting the difference," *Towards Data Science*, 20-Jun-2017. [Online]. Available: https://towardsdatascience.com/pytorch-vs-tensorflow-spotting-the-difference-25c75777377b. [Accessed: 25-Sep-2018].
[36] U. India, "Tensorflow or PyTorch : The Force is Strong with which One?," *Medium*, 24-Apr-2018. .
[37] PyTorch, *examples/regression at master · pytorch/examples*. 2019.
[38] "Vivado High-Level Synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. [Accessed: 16-Nov-2018].
[39] Vivado, "Vivado Design Suite," 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed: 29-Sep-2018].
[40] Xilinx, "AXI Reference Guide," vol. 13, p. 82, 2011.
[41] PyTorch, "torch.nn — PyTorch master documentation," 2019. [Online]. Available: https://pytorch.org/docs/0.4.1/nn.html#rnn. [Accessed: 16-Feb-2019].
[42] PyTorch, *pytorch/pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration*. 2019.
[43] "SampleRNN in PyTorch." [Online]. Available: http://deepsound.io/samplernn_pytorch.html. [Accessed: 14-Nov-2018].

APPENDIX B LOGBOOK SUMMARY SHEET



Week No.	Date	Comments, if applicable	Student's Signature	Supervisor's Signature
1	04/02/19		[illegible]	[illegible]
2	12/02/19		[illegible]	[illegible]
3	17/02/19		[illegible]	[illegible]
3	19/02/19		[illegible]	[illegible]
4	24/02/19		[illegible]	[illegible]
4	26/02/19		[illegible]	[illegible]
5	03/03/19		[illegible]	[illegible]
5	07/03/19		[illegible]	[illegible]
6	10/03/19		[illegible]	[illegible]

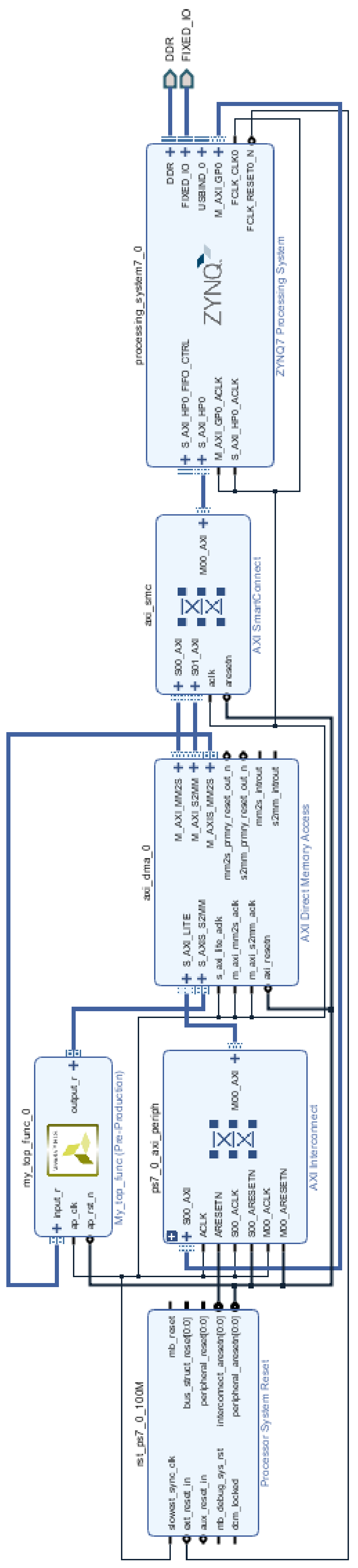
APPENDIX C STEPS TO PORT & TEST PYTORCH

1. Login as super user: `sudo su`
2. Update already installed packages: `apt-get update && apt-get upgrade`
3. Restart: `shutdown -r now`
4. Login as super user: `sudo su`
5. Go to root directory: `cd /`
6. Create a SWAP file of size 10 gigabytes: `dd if=/dev/zero of=swapfile bs=1M count=10000`
7. Make it compatible: `mkswap swapfile`
8. Start using the SWAP file created: `swapon swapfile`
9. Add new SWAP file to configuration file: `nano /etc/fstab`
    a. Keep a note of the location of the default SWAP file of size 1GB. We will need this later. (/var/swap/)
    b. Remove all other lines and add this: /swapfile none swap sw 0 0
10. Remove default SWAP file which comes with the OS: `rm -R /var/swap` (OR WHATEVER LOCATION WAS IN STEP 9)
11. Check if swap is active: `swapon -s`
12. Restart: `shutdown -r now`
13. Login as super user: `sudo su`
14. Provide permissions to use new SWAP file: `chmod 600 /swapfile`
15. Install PyTorch dependencies: `apt-get install libopenblas-dev cython3 libatlas-base-dev m4 libblas-dev cmake cython python3-dev python3-yaml`
16. Install other useful packages: `pip3 install --user pyyaml numpy typing tmux`
17. Restart: `shutdown -r now`
18. Login as super user: `sudo su`
19. Clone the PyTorch repository from GitHub: `git clone --recursive https://github.com/pytorch/pytorch`
20. Open the directory: `cd pytorch`
21. Obtain v0.4.1 of PyTorch: `git checkout tags/v0.4.1 -b build`
22. Update modules: `git submodule update --init --recursive`
23. Enter tmux (allows leaving the terminal but still running process): `tmux`
24. Set flag: `export NO_CUDA=1`
25. Set flag: `export NO_DISTRIBUTED=1`
26. Set flag: `export MAX_JOBS=1`
27. Build dependencies: `python3 setup.py build_deps`
28. Install PyTorch: `python3 setup.py develop`

Program to test:

```
import torch # Usually error here if not ported correctly
x = torch.randn(5,5) # Create a 5x5 random-valued tensor
y = torch.randn(5,5) # Create another 5x5 random-valued tensor
z = x + y # Perform the element-wise addition
print(z) # Output Result. Should be a 5x5 tensor (unique)
```

APPENDIX D GENERIC VIVADO BLOCK DESIGN



APPENDIX E SOFTWARE RE-IMPLMENTATION OF RNN LAYER

```
def softwareRNNLayer(rnn):
    # Pre processing
    counter = 0;
    for weight in rnn.parameters():
        if (counter == 0):
            weight_ih_0 = weight.data
            counter = counter + 1
        elif (counter == 1):
            weight_hh_0 = weight.data
            counter = counter + 1
        elif (counter == 2):
            bias_ih_0 = weight.data
            counter = counter + 1
        elif (counter == 3):
            bias_hh_0 = weight.data
            counter = counter + 1
        elif (counter == 4):
            weight_ih_1 = weight.data
            counter = counter + 1
        elif (counter == 5):
            weight_hh_1 = weight.data
            counter = counter + 1
        elif (counter == 6):
            bias_ih_1 = weight.data
            counter = counter + 1
        elif (counter == 7):
            bias_hh_1 = weight.data
            counter = counter + 1
    # Calculating Hidden Layer 1 output
    output = torch.tanh(torch.mm(weight_ih_0, input[0].t()).t() + bias_ih_0 +
torch.mm(weight_hh_0, h0[0].t()).t() + bias_hh_0) # Equation from the Documentation
    output = output.reshape(1,1,dim) # Post Processing for hidden layer 1
    # Calculating Hidden Layer 2 output
    output_1 = torch.tanh(torch.mm(weight_ih_1, output[0].t()).t() + bias_ih_1 +
torch.mm(weight_hh_1, h0[1].t()).t() + bias_hh_1)
    output_1 = output_1.reshape(1,1,dim)

    # Putting the above two results together
    hn = torch.cat((output, output_1), 0).reshape(2,1,128)

    print(output_1)
    return output_1, hn
```

APPENDIX F VIVADO HLS CODE FOR RNN LAYER

```
#include <ap_axi_sdata.h> // Used for inclusion of fixed-point datatype.
#include "hls_math.h" // Allows performing tanh in hardware.

/* The following are constants used to pull data out of the stream
 * If the program needs to be adapted for another dimension, only these
 * values must be altered.
 */
#define SIZE 128
#define OFST1 2*SIZE
#define OFST2 SIZE*SIZE
#define OFST3 3*SIZE
#define OFST4 OFST3+OFST2
#define OFST5 OFST4+OFST2
#define OFST6 OFST5+SIZE
#define OFST7 OFST6+SIZE
#define OFST8 OFST7+OFST2
#define OFST9 OFST8+OFST2
#define OFST10 OFST9+SIZE
#define TOT_OUTPUT OFST1
#define TOT_INPUT OFST10+SIZE

// Structure to hold stream items.
struct datatype {
  float data;
  bool last;
};

// Two fixed-point datatypes defined after analysing the nature of data being used.
typedef ap_fixed<21,5> stream_fixed_type;
typedef ap_fixed<18,2> fixed2;

// Top Function:
void rnn_larger(datatype input[TOT_INPUT], datatype output[TOT_OUTPUT]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // Using no protocol at block-level.
#pragma HLS INTERFACE axis port=input // Both I/O are AXI Streams.
#pragma HLS INTERFACE axis port=output

        // Variable Declaration
        int i, j, counter;
        stream_fixed_type x[SIZE], h0[2*SIZE];
        stream_fixed_type mult_temp[SIZE], mult_temp2[SIZE];
        stream_fixed_type w_ih[SIZE*SIZE], w_hh[SIZE*SIZE], b_ih[SIZE], b_hh[SIZE];
        fixed2 out1[OFST1] = {};
        static bool last[OFST1] = {};

        /* Array holds the last flag details. The last index is set to HIGH to
         * notify the CPU that the output values are sent completely.
         */
        last[OFST1-1] = true;

        /* The following for loops are pulling data out off the stream
         * to begin the calculation of the first hidden layer's output.
         */
        for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
                x[i] = input[i].data;
        }

        for (i = 0; i < OFST1; i++) {
#pragma HLS PIPELINE
                h0[i] = input[i+SIZE].data;
        }

        for (i = 0; i < OFST2; i++) {
#pragma HLS PIPELINE
                w_ih[i] = input[i+OFST3].data;
        }

        for (i = 0; i < OFST2; i++) {
```

```
#pragma HLS PIPELINE
		w_hh[i] = input[i+OFST4].data;
	}

	for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
		b_ih[i] = input[i+OFST5].data;
	}

	for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
		b_hh[i] = input[i+OFST6].data;
	}

	/* This for loop ensures that when the accelerator is called multiple
	 * times by the CPU, the values from the previous run are not accumulated.
	 * Done by simply setting values to 0.
	 */
	for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
		mult_temp[i] = 0;
		mult_temp2[i] = 0;
	}

	// Now that the data is all loaded, the matrix multiplications can be conducted.
	counter = 0;
	for (i = 0; i < SIZE; i++) {
		for (j = 0; j < SIZE; j++) {
#pragma HLS UNROLL factor=32 // Unrolling this loop 32 times.
			mult_temp[i] = mult_temp[i] + (w_ih[counter] * x[j]);
			mult_temp2[i] = mult_temp2[i] + (w_hh[counter] * h0[j]);
			counter++;
		}
	}

	/* Once the matrix multiplication is complete, the activation function can be applied.
	 * Note that here the bias is also added.
	 */
	for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
		out1[i] = tanh((float)(mult_temp[i] + b_ih[i] + mult_temp2[i] + b_hh[i]));
	}

	/* The following for loops are pulling data out off the stream
	 * to begin the calculation of the second hidden layer's output.
	 */
	for (i = 0; i < OFST2; i++) {
#pragma HLS PIPELINE
		w_ih[i] = input[i+OFST7].data;
	}

	for (i = 0; i < OFST2; i++) {
#pragma HLS PIPELINE
		w_hh[i] = input[i+OFST8].data;
	}

	for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
		b_ih[i] = input[i+OFST9].data;
	}

	for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
		b_hh[i] = input[i+OFST10].data;
	}

	/* Once again the array is cleared. Note that this is the same
	 * array as before used to save space rather than declaring
	 * four arrays.
	 */
	for (i = 0; i < SIZE; i++) {
```

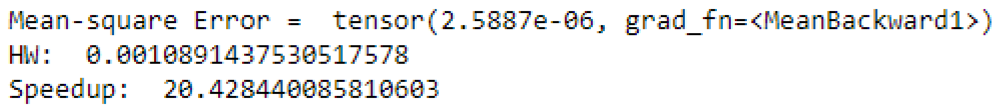
```
#pragma HLS PIPELINE
                mult_temp[i] = 0;
                mult_temp2[i] = 0;
        }

        /* Matrix multiplication again. Notice that the input x here is
         * here is actually the output of the first hidden layer.
         */
        counter = 0;
        for (i = 0; i < SIZE; i++) {
                for (j = 0; j < SIZE; j++) {
#pragma HLS UNROLL factor=32
                        mult_temp[i] = mult_temp[i] + (w_ih[counter] * out1[j]);
                        mult_temp2[i] = mult_temp2[i] + (w_hh[counter] * h0[j+SIZE]);
                        counter++;
                }
        }

        // Activation function again.
        for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
                out1[i+SIZE] = tanh((float)(mult_temp[i] + b_ih[i] + mult_temp2[i] + b_hh[i]));
        }

        // Sending the data back to the CPU.
        for (i = 0; i < OFST1; i++) {
#pragma HLS PIPELINE
                output[i].data = out1[i];
                output[i].last = last[i];
        }

}
```

APPENDIX G RNN LAYER PYTHON PROGRAM

```
import torch # PyTorch Library
import pynq.lib.dma # DMA Module
from pynq import Xlnk # Physical Memory Allocation Module
import numpy as np # NumPy Library for Arrays
from pynq import Overlay # Overlay Module
import time # Measuring execution time

overlay = Overlay('./rnn_new_test4.bit') # Loading the bitstream onto the FPGA
dma1 = overlay.axi_dma_0 # Referencing the DMA within the block design
xlnk = Xlnk() # Creating an instance of the Memory Manager
dim = 128 # Size of features
in_stream  =  xlnk.cma_array(shape=(3*dim+2*(dim*dim+dim*dim+dim+dim),1),  dtype=np.float32)  #
Allocating memory for input
out_stream = xlnk.cma_array(shape=(2*dim,1), dtype=np.float32) # Allocating memory for output

rnn  =  torch.nn.RNN(dim,  dim,  2)  #  Creating  an  RNN  Layer:  Mode  =  tanh,  Input_size  =  128,
Hidden_size = 128, Num_layers = 2
input  = torch.randn(1, 1, dim) # Generating a random input tensor
h0 = torch.randn(2, 1, dim) # Generating a random initial hidden state

# The following loop performs a forward pass on CPU
deltaT = 0 # Variable to accumulate time taken per forward pass
for i in range(5): # Forward pass done 5 times
    startTime = time.time() # Used to measure time
    output, hn = rnn(input, h0) # Forward pass
    endTime = time.time()
    deltaT = deltaT + (endTime - startTime) # Accumulate time per pass
print('SW: ', deltaT/5) # Display average execution time

# Pre-processing:

# Obtain weights and biases
counter = 0;
for weight in rnn.parameters():
    if (counter == 0):
        weight_ih_0 = weight.data
        counter = counter + 1
    elif (counter == 1):
        weight_hh_0 = weight.data
        counter = counter + 1
    elif (counter == 2):
        bias_ih_0 = weight.data
        counter = counter + 1
    elif (counter == 3):
        bias_hh_0 = weight.data
        counter = counter + 1
    elif (counter == 4):
        weight_ih_1 = weight.data
        counter = counter + 1
    elif (counter == 5):
        weight_hh_1 = weight.data
        counter = counter + 1
    elif (counter == 6):
        bias_ih_1 = weight.data
        counter = counter + 1
    elif (counter == 7):
        bias_hh_1 = weight.data
        counter = counter + 1

# Placing data into the Physical Memory
in_stream[:]         =         torch.cat((input[0].t(),         h0[0].t(),         h0[1].t(),
weight_ih_0.reshape(dim*dim,1),weight_hh_0.reshape(dim*dim,1),      bias_ih_0.reshape(dim,1),
bias_hh_0.reshape(dim,1),
weight_ih_1.reshape(dim*dim,1),weight_hh_1.reshape(dim*dim,1),bias_ih_1.reshape(dim,1),bias_h
h_1.reshape(dim,1)), 0)

# The following loop performs a forward pass in Hardware
deltaT_hw = 0 # Accumulation of time taken per forward pass
```

```
for i in range(5): # Hardware also is run 5 times
    startTime_hw = time.time() # Used for measuring time
    dma1.sendchannel.transfer(in_stream) # Provide the DMA the Physical Address of in_stream
    dma1.recvchannel.transfer(out_stream) # Provide the DMA the Physical Address of out_stream
    dma1.sendchannel.wait() # Wait for the transfer to complete
    dma1.recvchannel.wait()
    endTime_hw = time.time() # Used for measuring time
    deltaT_hw = deltaT_hw + (endTime_hw - startTime_hw) # Accumulate

# Post-processing:
hn_output = torch.tensor(out_stream).reshape(2,1,dim) # output of both hidden layers
output_hw = hn_output[1] # actual output

# Performances:
print('Mean-square Error = ', torch.mean((output - output_hw).pow(2)))
print('HW: ', deltaT_hw/5)
print('Speedup: ', (deltaT/5) / (deltaT_hw/5))
```

Results:



```
Mean-square Error =  tensor(2.5887e-06, grad_fn=<MeanBackward1>)
HW:  0.0010891437530517578
Speedup:  20.428440085810603
```

APPENDIX H STEPS TO PORT & TEST LIBROSA

All commands were executed as a super user.

Source: https://github.com/librosa/librosa/issues/757

1. Download LLVM 6.0.1 source code
    a. http://releases.llvm.org/download.html#6.0.1
    b. Unzip the file: `tar -xf llvm-6.0.1.src.tar.gz`
2. Create a directory for the build
    a. `mkdir llvm_build`
    b. `cd llvm_build`
3. Configure the LLVM build
    a. `cmake ~/Downloads/llvm-6.0.1.src -DLLVM_TARGETS_TO_BUILD="ARM" -DCMAKE_BUILD_TYPE="Release"`
4. Begin the build, make sure to use only 1 job as the PYNQ has limited RAM.
    a. `cmake –build . -- -j1`
5. Install once build finishes
    a. `cmake –build . --target install`
6. Install librosa 0.6.1
    a. `pip3 install librosa==0.6.1`
7. Install llvmlite 0.24.0 (This command will automatically uninstall the package which comes with librosa 0.6.1, which is what we want)
    a. `pip3 install llvmlite==0.24.0`
8. Install the correct Numba version (Otherwise error. This also uninstalls the numba package with librosa 0.6.1 which again is needed for the llvmlite version)
    a. `pip3 install numba==0.39.0`

Test:

```
python3
import librosa
# No error should pop up now.
```

APPENDIX I TOP-LEVEL PROFILING RESULTS

ncalls	tottime	percall	cumtime	percall	filename
1	0.001	0.001	545.452	545.452	/home/xilinx/samplernn-pytorch/trainer/plugins.py:157(epoch)
1	12.676	12.676	545.442	545.442	/home/xilinx/samplernn-pytorch/model.py:249(__call__)
86250/17250	6.944	0	523.407	0.03	/home/xilinx/pytorch/torch/nn/modules/module.py:471(__call__)
16000	8.001	0.001	501.484	0.031	/home/xilinx/samplernn-pytorch/model.py:167(forward)
50500	1.181	0	344.561	0.007	/home/xilinx/pytorch/torch/nn/utils/weight_norm.py:102(__call__)
50500	193.402	0.004	326.564	0.006	/home/xilinx/pytorch/torch/nn/utils/weight_norm.py:39(compute_weight)
50500	2.441	0	131.344	0.003	/home/xilinx/pytorch/torch/nn/modules/conv.py:174(forward)
50500	3.074	0	129.743	0.003	/home/xilinx/pytorch/torch/nn/utils/weight_norm.py:20(_norm)
49250	128.374	0.003	128.374	0.003	{built-in method conv1d}
49250	117.079	0.002	117.079	0.002	{method 'norm' of 'torch._C._TensorBase' objects}
1250	0.058	0	20.5	0.016	/home/xilinx/samplernn-pytorch/model.py:198(run_rnn)
1250	0.363	0	20.327	0.016	/home/xilinx/samplernn-pytorch/model.py:99(forward)
1250	1.77	0	16.816	0	{built-in method builtins.setattr}
1250	13.948	0	15.046	0	/home/xilinx/pytorch/torch/nn/modules/module.py:530(__setattr__)
1250	13.002	0	13.002	0	{method 'view' of 'torch._C._TensorBase' objects}
50521	0.41	0	9.024	0.007	/home/xilinx/samplernn-pytorch/nn.py:34(forward)
50500	0.225	0	8.276	0.007	/home/xilinx/pytorch/torch/nn/modules/rnn.py:153(forward)
1250	0.101	0	6.671	0.005	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:318(forward)
1250	0.063	0	6.47	0.005	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:259(forward)
1250	0.338	0	6.261	0.005	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:94(forward)
1250	0.422	0	5.551	0.002	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:131(forward)
2500	1.439	0.001	4.707	0.002	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:60(GRUCell)
2500	0.384	0	4.295	0	/home/xilinx/pytorch/torch/nn/functional.py:960(log_softmax)
5000	0.652	0	3.744	0	/home/xilinx/pytorch/torch/nn/modules/sparse.py:107(forward)
5000	2.697	0	3.673	0	{built-in method builtins.getattr}
170000	3.534	0	3.534	0	/home/xilinx/pytorch/torch/nn/modules/module.py:514(__getattr__)
16000	0.087	0	3.419	0.003	/home/xilinx/pytorch/torch/nn/modules/conv.py:555(forward)
111000	3.402	0	3.402	0	{method 'multinomial' of 'torch._C._TensorBase' objects}
285761	3.312	0.003	3.312	0.003	{built-in method conv_transpose1d}
16000	0.193	0	2.937	0	/home/xilinx/pytorch/torch/nn/functional.py:1036(embedding)
16000	2.749	0	2.749	0	{method 'permute' of 'torch._C._TensorBase' objects}

53000	2.745	0	2.745	0	{built-in method embedding}
16000	2.374	0	2.374	0	{method 'log_softmax' of 'torch._C._TensorBase' objects}
16000	0.154	0	2.144	0	/home/xilinx/pytorch/torch/nn/functional.py:1010(linear)
32000	0.396	0	2.113	0	/home/xilinx/pytorch/torch/nn/functional.py:635(relu)
16000	1.982	0	1.982	0	{method 'exp_' of 'torch._C._TensorBase' objects}
32000	1.719	0	1.719	0	{built-in method addmm}
16000	1.717	0	1.717	0	{built-in method relu}
16000	1.507	0	1.507	0	{method 'squeeze' of 'torch._C._TensorBase' objects}
32000	0.389	0	1.475	0	/home/xilinx/pytorch/torch/nn/functional.py:839(_get_softmax_dim)
133252	1.145	0	1.145	0	{method 'size' of 'torch._C._TensorBase' objects}
16000	1.085	0	1.086	0	{built-in method _warnings.warn}
20752	1.068	0	1.068	0	{method 'unsqueeze' of 'torch._C._TensorBase' objects}
11250	0.191	0	0.872	0	/home/xilinx/pytorch/torch/nn/modules/rnn.py:170(<genexpr>)
188502	0.69	0	0.69	0	{method 'values' of 'collections.OrderedDict' objects}
5000	0.076	0	0.633	0	/home/xilinx/pytorch/torch/nn/modules/module.py:731(parameters)
152751	0.598	0	0.598	0	{method 'get' of 'dict' objects}
11250	0.266	0	0.557	0	/home/xilinx/pytorch/torch/nn/modules/module.py:750(named_parameters)
105757	0.539	0	0.539	0	{built-in method builtins.isinstance}
5000	0.481	0	0.481	0	{method 'chunk' of 'torch._C._TensorBase' objects}
11250	0.317	0	0.465	0	/home/xilinx/pytorch/torch/nn/modules/container.py:155(__iter__)
83752	0.456	0	0.456	0	{method 'contiguous' of 'torch._C._TensorBase' objects}
6250	0.441	0	0.441	0	{built-in method sigmoid}
16002	0.407	0	0.407	0	{built-in method cat}
95286/92022	0.366	0	0.393	0	{built-in method builtins.len}
2500	0.018	0	0.352	0	/home/xilinx/pytorch/torch/nn/modules/rnn.py:231(all_weights)
1250	0.029	0	0.334	0	/home/xilinx/pytorch/torch/nn/modules/rnn.py:233(<listcomp>)
1250	0.252	0	0.252	0	{method 't' of 'torch._C._TensorBase' objects}
5000	0.246	0	0.246	0	{method 'dim' of 'torch._C._TensorBase' objects}
1250	0.179	0	0.241	0	/home/xilinx/samplernn-pytorch/utils.py:16(linear_dequantize)
75250	0.202	0	0.202	0	{built-in method tanh}
10000	0.107	0	0.181	0	{method 'add' of 'set' objects}
20000	0.108	0	0.156	0	/home/xilinx/pytorch/torch/tensor.py:384(__hash__)
2500	0.146	0	0.146	0	{method 'transpose' of 'torch._C._TensorBase' objects}
1250	0.055	0	0.11	0	/home/xilinx/pytorch/torch/nn/modules/rnn.py:120(check_forward_args)
16002	0.02	0	0.093	0	/home/xilinx/pytorch/torch/nn/modules/container.py:132(__getitem__)
1005	0.092	0	0.092	0	{built-in method builtins.iter}

1252	0.071	0	0.071	0	{method 'expand' of 'torch._C._TensorBase' objects}
1251	0.032	0	0.067	0	/home/xilinx/pytorch/torch/nn/modules/container.py:123(_get_abs_string_index)
1005	0.064	0	0.064	0	{method 'float' of 'torch._C._TensorBase' objects}
1250	0.036	0	0.06	0	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:231(AutogradRNN)
20000	0.048	0	0.048	0	{built-in method builtins.id}
10000	0.048	0	0.048	0	{method 'data_ptr' of 'torch._C._TensorBase' objects}
1250	0.027	0	0.036	0	/home/xilinx/pytorch/torch/nn/modules/container.py:152(__len__)
3264	0.024	0	0.033	0	/home/xilinx/pytorch/torch/nn/modules/rnn.py:141(check_hidden_size)
7500	0.024	0	0.024	0	{method 'append' of 'list' objects}
1250	0.02	0	0.023	0	/home/xilinx/pytorch/torch/nn/modules/module.py:795(named_children)
1250	0.017	0	0.022	0	/home/xilinx/pytorch/torch/backends/cudnn/__init__.py:82(is_acceptable)
1250	0.013	0	0.019	0	/home/xilinx/pytorch/torch/nn/backends/backend.py:7(__getattr__)
1250	0.018	0	0.018	0	{built-in method torch._C._jit_is_tracing}
1250	0.013	0	0.017	0	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:89(StackedRNN)
1250	0.012	0	0.012	0	{method 'detach' of 'torch._C._TensorBase' objects}
1	0.008	0	0.008	0	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:316(RNN)
1250	0.007	0	0.007	0	/home/xilinx/pytorch/torch/nn/modules/conv.py:436(_output_padding)
1250	0.007	0	0.007	0	{method 'items' of 'collections.OrderedDict' objects}
2500	0.007	0	0.007	0	/home/xilinx/pytorch/torch/nn/_functions/rnn.py:130(Recurrent)
1250	0	0	0.007	0.007	/usr/local/lib/python3.6/dist-packages/librosa/output.py:187(write_wav)
1	0.006	0	0.006	0	{built-in method torch._C._get_cudnn_enabled}
1250	0.003	0.003	0.004	0.004	/usr/local/lib/python3.6/dist-packages/librosa/util/utils.py:552(normalize)
1	0.003	0	0.003	0	{built-in method _operator.index}
1	0	0	0.002	0.002	/usr/lib/python3/dist-packages/scipy/io/wavfile.py:284(write)
1005	0	0	0.001	0	/usr/lib/python3.6/warnings.py:85(_showwarnmsg)
3	0	0	0.001	0	/usr/lib/python3.6/warnings.py:20(_showwarnmsg_impl)
3	0	0	0.001	0.001	/usr/lib/python3/dist-packages/scipy/io/wavfile.py:400(_array_tofile)
1	0	0	0.001	0.001	/usr/local/lib/python3.6/dist-packages/librosa/util/utils.py:110(valid_audio)
5	0.001	0	0.001	0	{method 'write' of '_io.BufferedWriter' objects}
1	0.001	0	0.001	0	{method 'reduce' of 'numpy.ufunc' objects}

APPENDIX J LINE PROFILING RESULTS

```
Timer unit: 1e-06 s
Total time: 5.36674 s
File: /home/xilinx/pytorch/torch/nn/_functions/rnn.py
Function: GRUCell at line 75

Hits         Time  Per Hit   % Time  Line Contents
==================================================
                                     @profile
                                     def GRUCell(input, hidden, w_ih, w_hh, b_ih=None, b_hh=None):
2500    1302883.0    521.2     24.3      gi = F.linear(input, w_ih, b_ih)
2500    1303676.0    521.5     24.3      gh = F.linear(hidden, w_hh, b_hh)

2500     318547.0    127.4      5.9      i_r, i_i, i_n = gi.chunk(3, 1)
2500     265156.0    106.1      4.9      h_r, h_i, h_n = gh.chunk(3, 1)

2500     580300.0    232.1     10.8      resetgate = torch.sigmoid(i_r + h_r)
2500     426801.0    170.7      8.0      inputgate = torch.sigmoid(i_i + h_i)
2500     603764.0    241.5     11.3      newgate = torch.tanh(i_n + resetgate * h_n)
2500     534629.0    213.9     10.0      hy = newgate + inputgate * (hidden - newgate)

2500      30982.0     12.4      0.6      return hy

Total time: 2.14771 s
File: /home/xilinx/pytorch/torch/nn/functional.py
Function: linear at line 1009

Hits         Time  Per Hit   % Time  Line Contents
==================================================
                                     @profile
                                     def linear(input, weight, bias=None):
                                         r"""
                                         Applies a linear transformation to the incoming data: :math:`y =
xA^T + b`.

                                         Shape:

                                             - Input: :math:`(N, *, in\_features)` where `*` means any number
of
                                               additional dimensions
                                             - Weight: :math:`(out\_features, in\_features)`
                                             - Bias: :math:`(out\_features)`
                                             - Output: :math:`(N, *, out\_features)`
                                         """
5000      73913.0     14.8      3.4      if input.dim() == 2 and bias is not None:
                                             # fused op is marginally faster
5000    2073802.0    414.8     96.6          return torch.addmm(bias, input, weight.t())

                                         output = input.matmul(weight.t())
                                         if bias is not None:
                                             output += bias
                                         return output

Total time: 129.681 s
File: /home/xilinx/pytorch/torch/nn/utils/weight_norm.py
Function: _norm at line 54

 Hits         Time  Per Hit   % Time  Line Contents
===================================================
                                      @profile
                                      def _norm(p, dim):
                                          """Computes the norm over all dimensions except dim"""
50514     565315.0     11.2      0.4      if dim is None:
                                              return p.norm()
50514     517167.0     10.2      0.4      elif dim == 0:
50514    1356034.0     26.8      1.0          output_size = (p.size(0),) + (1,) * (p.dim() - 1)
50514    127242022.0      2518.9        98.1                   return  p.contiguous().view(p.size(0),  -
1).norm(dim=1).view(*output_size)
                                          elif dim == p.dim() - 1:
                                              output_size = (1,) * (p.dim() - 1) + (p.size(-1),)
                                              return           p.contiguous().view(-1,           p.size(-
1)).norm(dim=0).view(*output_size)
                                          else:
                                              return _norm(p.transpose(0, dim), 0).transpose(0, dim)
```

```
Total time: 335.674 s
File: /home/xilinx/pytorch/torch/nn/utils/weight_norm.py
Function: compute_weight at line 73

 Hits         Time  Per Hit   % Time  Line Contents
===================================================
                                      @profile
                                      def compute_weight(self, module):
50507    3558188.0     70.4      1.1          g = getattr(module, self.name + '_g')
50507    2724949.0     54.0      0.8          v = getattr(module, self.name + '_v')
50507    1604996.0     31.8      0.5           if (v.size()[0] == 64 and v.size()[1] == 64 and v.size()[2]
== 1):
16252   25135848.0   1546.6      7.5              return v * (g / _norm(v, self.dim))
34255     726835.0     21.2      0.2         elif (v.size()[0] == 256 and v.size()[1] == 64 and v.size()[2]
== 1):
16001   65264167.0   4078.8     19.4              return v * (g / _norm(v, self.dim))
18254  236658769.0  12964.8     70.5          return v * (g / _norm(v, self.dim))
```

APPENDIX K SOFTWARE RE-IMPLMENTATION OF WEIGHT NORMALISATION

```
def softwareWeightNormalisation(v, g):
    # Pre-processing
    v_copy = v.reshape(v.size()[0]*v.size()[1]*v.size()[2],1) # Reshape the tensor, v, into a
row vector
    g_copy = g.reshape(g.size()[0],1) # Reshape tensor g as well into a

    # Initialise tensors
    norm = torch.zeros(g_copy.size()) # Create tensor with the same dimension as row vector g,
with all zeros
    square_sum = torch.zeros(g_copy.size())
    div = torch.zeros(g_copy.size())
    output = torch.zeros(v_copy.size()) # Create tensor to hold the output, values initialised
to zero

    # Nested loop to calculate the norm and the division of g with the norm
    counter = 0 # Used to obtain value of v in order
    for i in range(v.size()[0]):
        for j in range(v.size()[1]*v.size()[2]):
            square_sum[i] = square_sum[i] +  (v_copy[counter] * v_copy[counter]) # Square v
and accumulate
            counter = counter + 1 # Each iteration of inner loop, continue to obtain new values
of v
        norm[i] = torch.sqrt(square_sum[i]) # Once sum of squares has been completed, perform
square-root
        div[i] = g_copy[i] / norm[i] # Divide

    # Nested loop to calcuate the multiplication of v with (g/norm(v))
    counter = 0
    for i in range(v.size()[0]):
        for j in range(v.size()[1]):
            for k in range(v.size()[2]):
                output[counter] = v_copy[counter] * div[i]
                counter = counter + 1

    # Post-processing
    return output.reshape(v.size())
```

APPENDIX L VIVADO HLS CODE FOR GRUCELL

```
#include <ap_axi_sdata.h> // Needed for Fixed-point datatype

// Constants
#define SIZE 64
#define W_SIZE 192*SIZE
#define B_SIZE 3*SIZE
#define OFST1 2*SIZE
#define OFST2 OFST1+W_SIZE
#define OFST3 OFST2+W_SIZE
#define OFST4 OFST3+B_SIZE
#define TOT_INPUT OFST4+B_SIZE
#define TOT_OUTPUT B_SIZE

// Structure for stream items
struct datatype {
  float data;
  bool last;
};

typedef ap_fixed<32,16> stream_fixed_type; // Fixed-point datatype of FIX_32_16

void sampleRNN_GRU(datatype input[TOT_INPUT], datatype output[2*TOT_OUTPUT]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // No block-level protocol
#pragma HLS INTERFACE axis port=input // Declaring I/O as stream
#pragma HLS INTERFACE axis port=output

        // Variables
        int i, j, counter;
        stream_fixed_type x[SIZE], h0[SIZE], temp1;
        stream_fixed_type      mult_temp[TOT_OUTPUT],      out1[2*TOT_OUTPUT]      =      {},
mult_temp2[TOT_OUTPUT];
        stream_fixed_type w_ih[W_SIZE], b_ih[B_SIZE], w_hh[W_SIZE],  b_hh[B_SIZE];
        bool last[2*TOT_OUTPUT] = {};
        last[2*TOT_OUTPUT-1] = true; // Setting the last flag HIGH for the last element

        // The following loops are segregating the data from the stream into their respective
variables
        for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
                x[i] = input[i].data;
        }

        for (i = 0; i < SIZE; i++) {
#pragma HLS PIPELINE
                h0[i] = input[i+SIZE].data;
        }

        for (i = 0; i < W_SIZE; i++) {
#pragma HLS PIPELINE
                w_ih[i] = input[i+OFST1].data;
        }

        for (i = 0; i < W_SIZE; i++) {
#pragma HLS PIPELINE
                w_hh[i] = input[i+OFST2].data;
        }

        for (i = 0; i < B_SIZE; i++) {
#pragma HLS PIPELINE
                b_ih[i] = input[i+OFST3].data;
        }

        for (i = 0; i < B_SIZE; i++) {
#pragma HLS PIPELINE
                b_hh[i] = input[i+OFST4].data;
        }

        // Loop performing 2 matrix multiplications and vector additions
        counter = 0;
        for (i = 0; i < TOT_OUTPUT; i++) {
                mult_temp[i] = 0; // Make sure the values are zero initially
                mult_temp2[i] = 0;
                for (j = 0; j < SIZE; j++) {
#pragma HLS UNROLL factor=16 // Matrix multiplication
                        mult_temp[i] = mult_temp[i] + (w_ih[counter] * x[j]);
                        mult_temp2[i] = mult_temp2[i] + (w_hh[counter] * h0[j]);
                        counter++;
                }
                out1[i] = mult_temp[i] + b_ih[i]; // Vector addition
                out1[i+TOT_OUTPUT] = mult_temp2[i] + b_hh[i];
        }

        // Send output to the stream
        for (i = 0; i < 2*TOT_OUTPUT; i++) {
#pragma HLS PIPELINE
                output[i].data = out1[i];
                output[i].last = last[i];
        }

}
```

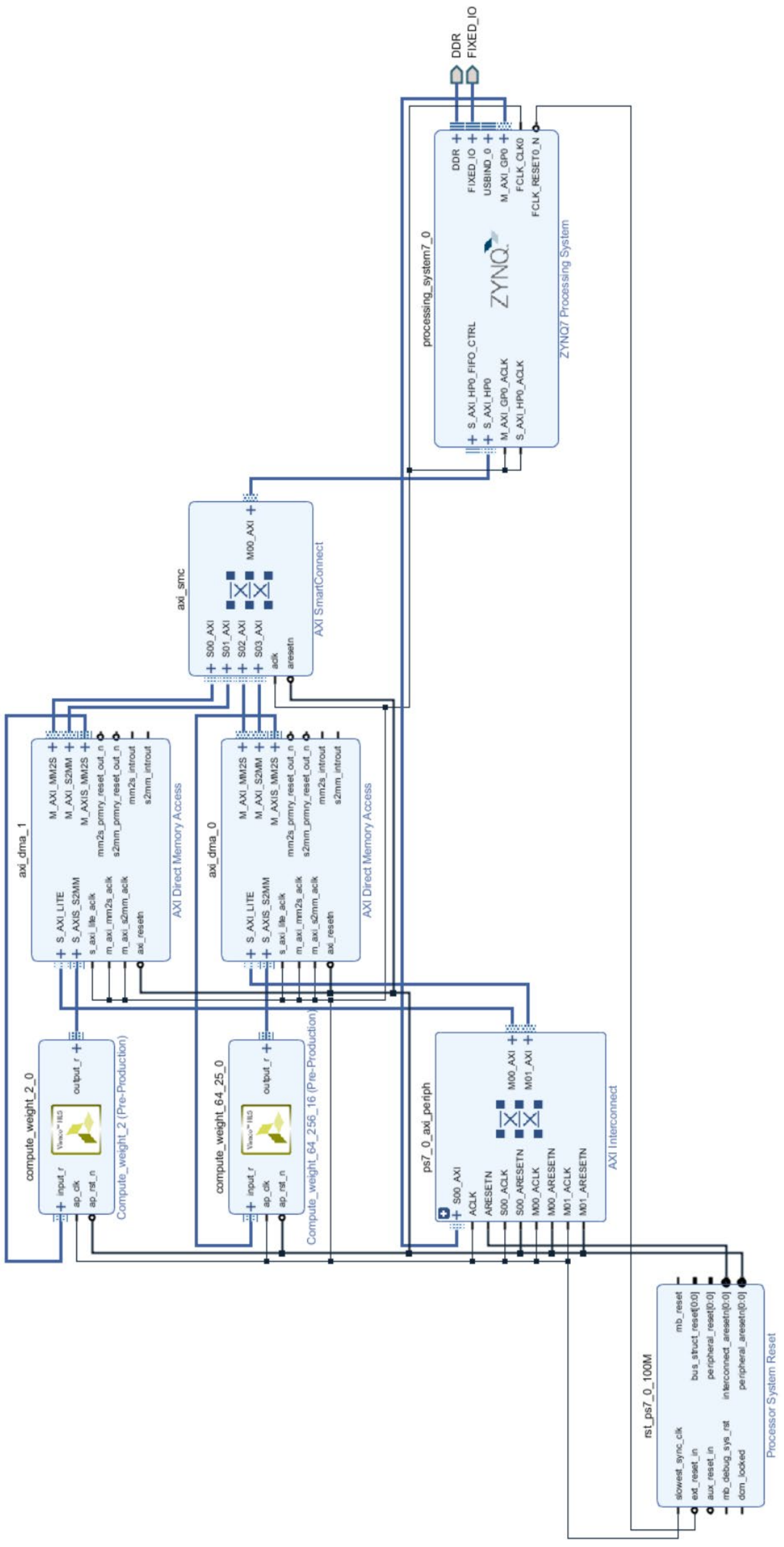
APPENDIX M VIVADO HLS CODE FOR WEIGHT NORMALISATION (64X64X1)

```
#include <ap_axi_sdata.h> // Needed for Fixed-point datatype.
#include "hls_math.h" // Needed for the square-root function.

// Constant definitions. Changing these allows the modification of the dimensions.
#define G_SIZE                  64
#define V_LAYERS                64
#define V_ROWS                  64
#define V_COL                   1
#define V_SIZE                  V_LAYERS*V_ROWS*V_COL
#define TOT_INPUT               G_SIZE+V_SIZE

// Structure to bring in items from the stream.
struct datatype {
  float data;
  bool last;
};

typedef ap_fixed<32,16> fixed2; // Fixed-point datatype FIX_32_16

void compute_weight_2(datatype input[TOT_INPUT], datatype output[V_SIZE]) {
#pragma HLS INTERFACE ap_ctrl_none port=return // No block-level protocol
#pragma HLS INTERFACE axis port=input // Declaring the I/O as stream
#pragma HLS INTERFACE axis port=output

        // Variable Declaration
        fixed2 norm[G_SIZE] = {}, v_in[V_SIZE], output_temp[V_SIZE];
        int i, j, k, counter, counter2;
        bool last[V_SIZE] = {};
        last[V_SIZE-1] = true; // Pre-setting the last variable's last element as HIGH.

        // Bringing in the input v.
        for (i = 0; i < V_SIZE; i++) {
#pragma HLS PIPELINE
                v_in[i] = input[i].data;
        }

        // Similar design to the software version created.
        counter = 0;
        counter2 = 0;
        for (i = 0; i < G_SIZE; i++) {
                norm[i] = 0; // Making sure it is zero before accumulating
                for (j = 0; j < V_ROWS*V_COL; j++) {
#pragma HLS UNROLL factor=16
                        norm[i] = norm[i] + (v_in[i] * v_in[i]);
                        counter++;
                }
                norm[i] = ((fixed2)input[V_SIZE+i].data) / hls::sqrt(norm[i]); // Using fixed-
point square-root
                for (j = 0; j < V_ROWS*V_COL; j++) {
/* Since this accelerator is used in two scenarios, when it is alone, keep UNROLL factor at
32, otherwise 16.
 * Done since there is lack of space.
 */
#pragma HLS UNROLL factor=32
                        output_temp[counter2] = v_in[counter2] * norm[i];
                        counter2++;
                }
        }

        // Sending the data back out through the stream.
        for (i = 0; i < V_SIZE; i++) {
#pragma HLS PIPELINE
                output[i].data = output_temp[i];
                output[i].last = last[i];
        }

}
```

APPENDIX N VIVADO HLS CODE FOR WEIGHT NORMALISATION (256X64X1)

```
/* Same as the code for v dimension: 64x64x1 & g dimension: 64x1x1.
 * But now the Constants are set to v dimension: 256x641 & g dimension: 256x1x1.
 * The UNROLLING factors have also changed.
 */

#include <ap_axi_sdata.h>
#include "hls_math.h"

#define G_SIZE                  256
#define V_LAYERS                256
#define V_ROWS                  64
#define V_COL                   1
#define V_SIZE                  V_LAYERS*V_ROWS*V_COL
#define TOT_INPUT               G_SIZE+V_SIZE

struct datatype {
  float data;
  bool last;
};

typedef ap_fixed<32,16> fixed2;

void compute_weight_64_256_16(datatype input[TOT_INPUT], datatype output[V_SIZE]) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE axis port=input
#pragma HLS INTERFACE axis port=output


        fixed2 norm[G_SIZE] = {}, v_in[V_SIZE], output_temp[V_SIZE];
        int i, j, k, counter, counter2;
        bool last[V_SIZE] = {};
        last[V_SIZE-1] = true;

        for (i = 0; i < V_SIZE; i++) {
#pragma HLS PIPELINE
                v_in[i] = input[i].data;
        }

        counter = 0;
        counter2 = 0;
        for (i = 0; i < G_SIZE; i++) {
                norm[i] = 0;
                for (j = 0; j < V_ROWS*V_COL; j++) {
#pragma HLS UNROLL factor=16 // Same as the other design
                        norm[i] = norm[i] + (v_in[i] * v_in[i]);
                        counter++;
                }
                norm[i] = ((fixed2)input[V_SIZE+i].data) / hls::sqrt(norm[i]);
                for (j = 0; j < V_ROWS*V_COL; j++) {
#pragma HLS UNROLL factor=64 // Unrolled completely
                        output_temp[counter2] = v_in[counter2] * norm[i];
                        counter2++;
                }
        }

        for (i = 0; i < V_SIZE; i++) {
#pragma HLS PIPELINE
                output[i].data = output_temp[i];
                output[i].last = last[i];
        }

}
```

APPENDIX O VIVADO 2018.2 COMBINATION ACCELERATOR WEIGHT NORMALISATION BLOCK DESIGN



DDR
FIXED_IO
processing_system7_0
ZYNQ7 Processing System
axi_smc
AXI SmartConnect
axi_dma_1
axi_dma_0
AXI Direct Memory Access
compute_weight_2_0
Compute_weight_2 (Pre-Production)
compute_weight_64_25_0
Compute_weight_64_256_16 (Pre-Production)
ps7_0_axi_periph
AXI Interconnect
rst_ps7_0_100M
Processor System Reset