

Hardware Acceleration of Linear Recurrent Units

by

Francisco Ayala Le Brun

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday August 19, 2024 at 9:30 AM.

Student number:	5074479	
Project duration:	November 20, 2023 – August 19, 2024	
Thesis committee:	Dr. Przemysław Pawełczak,	TU Delft, Supervisor
	Dr. Charlotte Frenkel,	TU Delft, Daily Supervisor
	Dr. João Sacramento,	Google

An electronic version of this thesis is available at
<http://repository.tudelft.nl/>.



Abstract

State-space models (SSMs) combine attention-like parallelization with RNN-like inference efficiency, using internal states with linear update and output functions, similar to RNNs but without non-linearities in the update function. Linear Recurrent Units (LRUs), a type of SSM, are well-suited to keyword spotting due to their ability to handle long-range dependencies. However, hardware acceleration for LRUs remains unexplored and presents challenges due to high hardware cost components such as the GELU, LayerNorm and complex multiplication. This work modifies the LRU model architecture to enable a more efficient hardware implementation and designs an accelerator tailored to the modified architecture. We propose the GRELU, a new activation function well-suited to inference on hardware. The modified model architecture achieved an accuracy of 95.5% @ 35 keywords on the Google Speech Commands dataset. The accelerator’s vector unit supports complex operations and reductions, with respective overheads of only 19.1% and 27.2% over basic operations. Our results demonstrate the proposed hardware accelerator’s efficiency and effectiveness at keyword spotting applications.

Contents

1	Introduction	1
2	Background	5
2.1	Recurrent Neural Networks	5
2.2	State-Space Models	6
2.3	Quantization	8
2.4	Systolic Matrix Multiplication.	10
3	Hardware-Software Co-Design	13
3.1	Network Architecture	14
3.2	GReLU	15
3.3	GLU Activation Function	16
3.4	LRU Simplification	17
3.5	Matrix Multiplication Quantization	18
4	Hardware Architecture	19
4.1	Overview	19
4.2	Vector Unit	22
4.3	Matrix Unit	24
5	Results	27
5.1	Experimental Setup	27
5.2	Design Space Exploration	29
5.3	Resource Utilization	29
5.4	Hardware Architecture Characteristics	32
5.4.1	Power Usage by Component	32
5.4.2	Special Function Overhead	33
5.4.3	Approach to LayerNorm	33
5.4.4	Hardware-friendly Activation Function	34
5.5	Comparison with other works.	34
6	Conclusion	37
	Appendices	38
A	Model Training Parameters	39

1

Introduction

In recent years, the transformer architecture [62] has found widespread use in sequence modelling tasks, such as classification based on video, audio, or text sequences. However, this has been tied to an exponential increase in compute demand [58]. This is partly due to a transformer’s complexity scaling quadratically in relation to sequence length [62]. In contrast, traditional architectures such as Recurrent Neural Networks (RNNs) offer linear scaling at the cost of temporal stability and slower training [35].

Neural architectures under the state-space model (SSM) umbrella address the aforementioned limitations by offering attention-like training parallelization together with RNN-like inference efficiency. SSMs model sequences based on an internal state, and linear update and output functions [25]. It is therefore in principle very similar to an RNN, with the main dissimilarity being the lack of a non-linearity in the update function. At the same time, it inherits similar weaknesses to the transformer architecture, such as an inability to perform state tracking [47]. One architecture under the SSM umbrella is the linear recurrent unit (LRU). This architecture uses the RNN as a starting point to arrive at an SSM architecture [51]. Nonlinearities in the hidden state are removed, with stacking and an added gated linear unit (GLU) used to maintain expressiveness in the network. See Section 2.2 for more details.

Several hardware accelerators have been developed for RNN and RNN-like network architectures. Spartus [17] is a field-programmable gate array (FPGA)-based long short-term memory (LSTM) accelerator with support

for temporal and spatial sparsity. Laika [20] is an application-specific integrated circuit (ASIC)-based LSTM accelerator focusing on very low power consumption. OCEAN [9] is an RNN accelerator with support for on-chip learning. ReckOn is a spiking RNN (sRNN) accelerator which implements on-chip learning with minimal hardware overhead [16].

Following the recent introduction of the LRU architecture, hardware design based on its characteristics is still at a nascent stage. This architecture has been found to have significant advantages in performance compared to RNNs, especially in tasks with long-range dependencies [51]. Additionally, due to its capability for training parallelization, the LRU may facilitate the deployment of more complex models on hardware accelerators. Therefore, the research question of this thesis is formulated as follows:

To what extent can the LRU be co-optimized with hardware to allow for power-efficient and capable sequence modelling?

In order to answer this question, we focus on keyword spotting with the LRU. In keyword spotting, spoken words are classified based on an audio recording. The reason why this task is a good fit for hardware and the LRU is two-fold. Firstly, high power efficiency (over 16 GOP/s/W for FPGA [59]) is desirable in edge inference [53], which dedicated hardware can more easily meet. Secondly, keyword spotting can present long sequence lengths (10k+ elements [57]), due to sampling frequency or simply recording length. Due to its aforementioned long-range performance characteristics, this second requirement is well suited to the LRU.

As part of our process, we first investigate the impact of quantization on accuracy and identify non-essential components of the network for the given task. Based on this optimized architecture and the unique computational requirements of the LRU, we propose an accelerator containing a vector unit with support for element-wise operations, complex multiplication and reductions with a high degree of hardware reuse. This allows for inference on real-time unprocessed speech data with sub-60 microsecond latency per sample. The main contributions of this work are:

- the **GRELU**, a GELU-RELU activation function hybrid well suited to inference in hardware,
- LRU-specific vector unit¹, allowing for the implementation of LRU operations via efficient hardware reuse.

¹Proven in an end-to-end accelerator.

The accelerator achieves an accuracy of 95.5% @ 35 keywords on the 16kHz 35-way Google Speech Commands dataset. Meanwhile, hardware overhead for complex operations is only 19.1% and 27.2% for reductions, showcasing the vector unit’s hardware reuse.

This thesis is organized into a series of chapters. **Chapter 2** starts by presenting pre-requisite information on both LRU- and hardware-related concepts. Then, **Chapter 3** covers hardware-algorithm co-design with the LRU neural architecture. Motivated by these optimizations, **Chapter 4** describes the architecture of the accelerator and its components. In **Chapter 5**, software and hardware results are presented and discussed. Finally, the thesis is concluded in **Chapter 6**, summarizing an answer to our main research question and presenting an outlook on future work.

2

Background

2.1. Recurrent Neural Networks

In traditional feed-forward neural networks (FNNs), information flows in a single direction, while avoiding any cycles or loops [72]. In contrast, recurrent neural networks (RNNs) are a class of neural architectures where the same network weights are repeatedly applied to some time-varying input. At each time step, the network produces both an output and a hidden state. The output depends on both the input and the hidden state, which is why the network is considered recurrent.

An advantage of an RNN over an FNN is the ability to efficiently handle temporally long inputs. The hidden state acts as memory for the RNN, and its output being defined on a per-time-step basis means real-time usage is feasible. In contrast, to achieve a similar effect with an FNN, it would be necessary to buffer each sequence in order to approximate the effects of memory, precluding real-time usage and introducing memory overhead. Alternatively, if an FNN with a small input size is used, and is applied to a window of the input, then this causes the network to ignore medium- and long-range time dependencies [23].

The RNN architecture was first introduced in [14], now known as the vanilla or Elman RNN. For input x_t , the hidden state h_t is defined as in [48]:

$$\mathbf{h}_t = \sigma(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}), \quad (2.1)$$

where σ is a nonlinear activation function, while $\mathbf{U} \in \mathbb{R}^{m \times n}$ and $\mathbf{W} \in \mathbb{R}^{m \times m}$

are the weight matrices for the input $\mathbf{x}_t \in \mathbb{R}^n$ and hidden state $\mathbf{h}_t \in \mathbb{R}^m$, respectively, at timestep t .

As shown in Equation 2.1, the $\mathbf{W}$ matrix is multiplied with each hidden state successively in order to arrive at a given timestep. In backpropagation through time (BPTT), a gradient is then carried through each hidden state from last to first [66]. The derivative of Equation 2.1 includes the matrix transpose $\mathbf{W}^T$. Therefore gradient calculation involves repeated application of $\mathbf{W}^T$. If the eigenvalues of $\mathbf{W}$ are greater than one, then the gradient will grow exponentially. Conversely, if eigenvalues are less than one, then the gradient will shrink exponentially. This is why the vanilla RNN suffers from the exploding/vanishing gradient problem, where gradients either grow exponentially and cause numerical instability (exploding gradients) or shrink to near zero and impede learning (vanishing gradients) [3].

Additionally, due to the non-linear effect of the activation function σ , it is not possible to parallelize training of RNNs. Hidden states must be evaluated in sequence [23].

2.2. State-Space Models

State-space models are defined as mathematical representations of dynamic systems and have recently been the subject of interest as alternatives to traditional RNNs and attention-based models [24]. They are defined by a state update ($\mathbf{h}_t$) and output ($\mathbf{y}_t$) equations:

$$\mathbf{h}_t = \mathbf{A}\mathbf{h}_{t-1} + \mathbf{B}\mathbf{x}_t, \quad (2.2)$$

$$\mathbf{y}_t = \mathbf{C}\mathbf{h}_t + \mathbf{D}\mathbf{x}_t, \quad (2.3)$$

where $\mathbf{h}_t \in \mathbb{R}^m$ is the state vector, $\mathbf{x}_t \in \mathbb{R}^k$ is the input vector, t the timestep, and $\mathbf{A} \in \mathbb{R}^{m \times m}$, $\mathbf{B} \in \mathbb{R}^{m \times k}$, $\mathbf{C} \in \mathbb{R}^{k \times m}$, $\mathbf{D} \in \mathbb{R}^{k \times k}$ are weight matrices.

SSMs offer the following advantages compared to traditional RNNs:

- *Parallelization.* Due to the use of linear equations for the state update, sequential inputs can be evaluated in parallel. This is possible via the use of a prefix sum [6, 55], which computes the sums of all possible prefixes for a given sequence. For example, the prefix sum of a sequence $[a_1, a_2, \dots, a_n]$ is $[a_1, a_1 + a_2, \dots, a_1 + a_2 + \dots + a_n]$. See Figure 2.1 for an example on how a prefix sum may be computed in parallel.
- *Efficient inference.* Sequential evaluation of inputs is still possible, resulting in a high degree of data locality and real-time operation for streaming use cases.

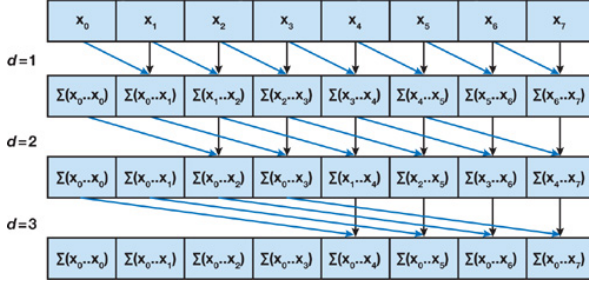


Figure 2.1: Example of algorithm for parallel computation of prefix sum [50].

- *Temporal stability.* The state update equation can be defined such that temporal stability is preserved. High-order polynomial projection operators (HiPPO [26]) matrices, used in S4 [25], dynamically project functions onto a basis of polynomials, ensuring recent information is emphasized, which helps maintain stability over time. Another example is exponential parameterization in the LRU [51], which involves parameterizing of the recurrent matrix $\mathbf{A}$ with a complex exponential to ensure the hidden state's norm does not grow much larger than one over time (see Equation 2.6).

Recent deep learning work has explored the use of SSMs for sequence modeling. We refer the reader to [24] for an overview of existing models. We now focus on one of these models, the linear recurrent unit (LRU) due to its simplicity and potential for online learning [76] (Online learning not explored in this work).

Linear Recurrent Unit

The LRU [51] is an SSM which incorporates concepts from both SSMs and RNNs. It is defined by the following equations:

$$\mathbf{h}_t = \lambda \odot \mathbf{h}_{t-1} + \gamma \odot \mathbf{B}\mathbf{x}_t \quad (2.4)$$

$$\mathbf{y}_t = \text{Re}[\mathbf{C}\mathbf{h}_t] + \mathbf{D}\mathbf{x}_t, \quad (2.5)$$

here $\lambda, \gamma \in \mathbb{C}^m$, $\mathbf{B} \in \mathbb{C}^{k \times m}$, $\mathbf{C} \in \mathbb{C}^{m \times k}$, and $\mathbf{D} \in \mathbb{R}^{k \times k}$ are parameters, while $\mathbf{h} \in \mathbb{C}^m$, and $\mathbf{y}, \mathbf{x} \in \mathbb{R}^k$ are states or activation vectors. $\odot$ represents element-wise multiplication. γ is a learnable normalization parameter. In order to ensure temporal stability, λ is parameterized as

$$\lambda = e^{-e^{\nu \log} + i e^{\theta \log}}, \quad (2.6)$$

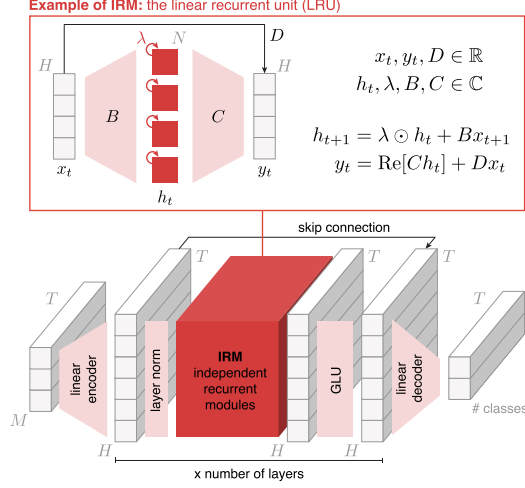


Figure 2.2: Illustration of LRU within a complete neural architecture [76]. See Section 2.2 for definition of the parameters in the figure.

where v^{log} and θ^{log} are learnable parameters. This ensures the magnitude of λ remains below 1, therefore ensuring temporal stability [76]. See Figure 2.2 for an illustration.

Gated Linear Unit

The architecture integrating the LRU as shown in Figure 2.2 makes use of the gated linear unit (GLU), defined as:

$$GLU(\mathbf{x}) = (\mathbf{W}\mathbf{x} + \mathbf{b}) \odot \sigma(\mathbf{V}\mathbf{x} + \mathbf{c}), \quad (2.7)$$

where σ is the sigmoid function, $\odot$ the element-wise product, $\mathbf{x}$ an input vector, and $\mathbf{W}$, $\mathbf{b}$, $\mathbf{V}$, and $\mathbf{c}$ learnable parameters.

2.3. Quantization

Most neural network training is done using floating point numbers. However, the bit width of numerical types used in an accelerator's computations can have severe implications on bandwidth, storage and resource utilization requirements. Hence, it is often beneficial to use lower-precision numeric formats where possible. The process of mapping "full-precision" 32-bit IEEE floating point numbers to lower-precision floating or fixed-point numbers is called quantization [27].

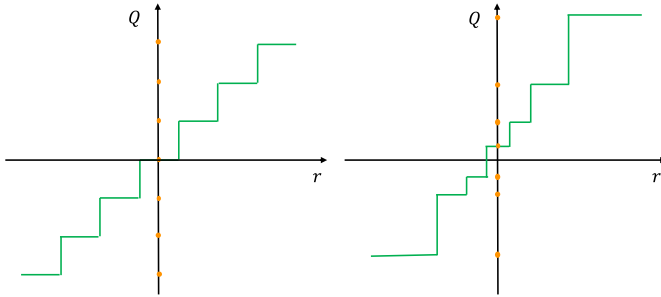


Figure 2.3: Example of uniform (left) and non-uniform (right) quantization functions [19]. In this figure, r represents the unquantized input, and Q the quantized output. The yellow dots are the available quantization steps. Note that the output (green line) always lands on an available quantization step. Uniform quantization spaces out these quantization steps evenly, while non-uniform quantization does not.

There are several types of quantization. These include uniform quantization [28, 75, 18] and non-uniform quantization [33, 63, 71, 40, 70]. See Figure 2.3 for visual examples. When performing uniform quantization, values are mapped to equally spaced points, such as when using fixed-point values. On the other hand, non-uniform quantization maps values non-uniformly. For example, quantization to a lower precision floating-point format is a form of non-uniform quantization, since floating-point numbers have greater resolution as they approach zero. In the case of neural networks, non-uniform quantization may be advantageous if it better fits the distribution of weights or activations [37].

Another important aspect to consider when performing quantization is how quantization is handled during training. Three main methodologies exist:

- Post-training quantization (PTQ) [42, 65]. Training is performed at full-precision and network weights are only statically quantized afterwards. This approach can be useful when non-quantized pre-trained weights are available, as a set of weights trained for full precision may be used as-is. However, this often results in a performance penalty compared to full-precision operation.
- Quantization-aware training (QAT) [36, 41, 73] quantizes forward activation values and weights during training, while preserving full precision in back-propagated gradients. Replacing the gradient of a quantization function in this way is called the straight-through estimator

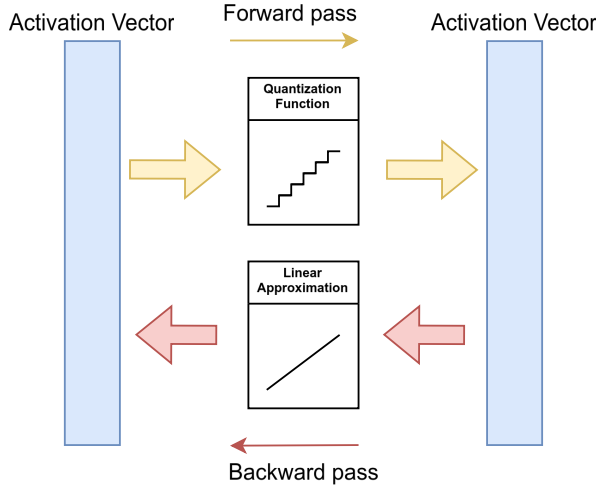


Figure 2.4: Graphical representation of the straight-through estimator.

(STE) [4]. See Figure 2.4 for a graphical representation. Since quantization is accounted for during training, the gap with full-precision operation is typically much lower compared to PTQ.

- Quantized training (QT) [74]. Quantization is applied to all values, including back-propagated gradients. Since no full-precision operations are performed during inference or training, this approach is most amenable to on-chip learning. However, such training is more difficult to achieve due to issues such as numerical instability and a lack of dynamic range.

2.4. Systolic Matrix Multiplication

First introduced in [39], systolic architectures make use of regular, rhythmic data flow between processing elements (PEs), their basic building block. Each PE performs computation, with inputs and outputs wired only to its neighbouring PEs. Such architectures are characterized by high throughput and low propagation delay due to the lack of global data transfers. However, due to the need to propagate signals over time they may suffer from high latency compared to combinatorial or sequential designs.

In the past decade, systolic architectures have been used in several com-

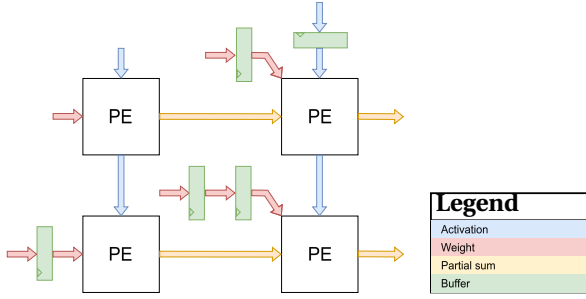


Figure 2.5: Example of a 2x2 systolic array for matrix-vector multiplication. Each processing element (PE) takes as inputs one activation (blue arrow) and one weight (red arrow), and one partial sum (yellow arrow, assumed zero for first column). The product of the weight and activation is added with the partial sum and output as a new partial sum, as well as passes the activation downwards. PE operations take one cycle between input and output in this example. Buffers (green) are inserted in order to ensure products are added to the correct partial sums. The final set of partial sums represents the result of the matrix-vector multiplication.

mercial deep learning accelerators [10]. See Figure 2.5 for an illustration of a systolic array for matrix-vector multiplication. One such accelerator is Google’s TPU [34]. The first version of this accelerator uses a 256×256 systolic array to perform matrix multiplication. Latency is substantial and proportional to the product of PE rows and columns. To address this, an accumulator memory is used in order to accommodate for batch sizes on the order of hundreds of elements. This allows utilization to remain high despite the latency.

The LRU model in this work does not make use of convolutional operators. Thus the main operation to be accelerated is not matrix multiplication but matrix-vector multiplication. However, a systolic architecture is still useful for matrix-vector multiplication due to the reuse of vector values, and in order to achieve a high frequency.

3

Hardware-Software Co-Design

We first consider our selected task and its requirements. In order to perform inference on unprocessed 16kHz samples, an accelerator must be able to process a new input vector every $60\text{ }\mu\text{s}$. An additional metric to improve is power consumption, with the goal of better efficiency relative to comparable accelerators. Therefore, our optimizations focus on the following points:

- simplifying the complexity and number of operations performed by the model,
- reducing the memory requirements of the network weights.

Task and Dataset

The inherent temporal stability of the LRU makes it well suited to sequence modeling tasks with long sequence lengths. An example of such a task is keyword spotting without length-reducing pre-processing, such as Mel-Frequency Cepstral Coefficients (MFCC). Length-reducing pre-processing refers to techniques that transform the input data into a more compact form, often by reducing the number of time steps while retaining essential information. MFCC is a popular method in speech processing that extracts features representing the short-term power spectrum of sound.

We choose to operate on the Google Speech Commands (GSC) dataset. This dataset presents an immediately practical use case for the model: keyword spotting. Additionally, we have the ability to operate on raw audio data,

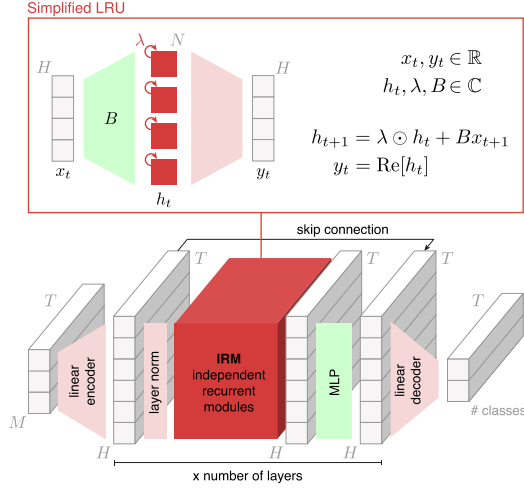


Figure 3.1: Modified LRU architecture. Adapted from Figure 2.2. Quantized blocks are underlined in green. We used an MLP in place of a GLU and simplified several components out of the LRU’s output function.

with 16,000 KHz 1-second audio samples typical in the dataset. This property allows the LRU to showcase its strengths on long-range dependencies. We perform 35-way keyword classification, taking the last output of the network as the result. Only the last output is taken as this represents the time when the hidden state contains the full context of the input.

3.1. Network Architecture

The baseline architecture follows what is proposed in [51]. This consists of a linear decoder followed by several layers of: layer normalization, a linear recurrent unit (See Section 2.2), a gated linear unit (GLU) [11], and a linear decoder with a skip connection to the layer norm. Finally, a linear decoder and argmax is used to perform classification. The dimensions of the network follow [55] with six layers, 192 hidden dimensions, and 128 output dimensions. This configuration achieves a baseline accuracy of 91.3%.

The findings in this section build upon the baseline architecture and result in the configuration shown in Figure 3.1 with the parameters summarized in Appendix A.

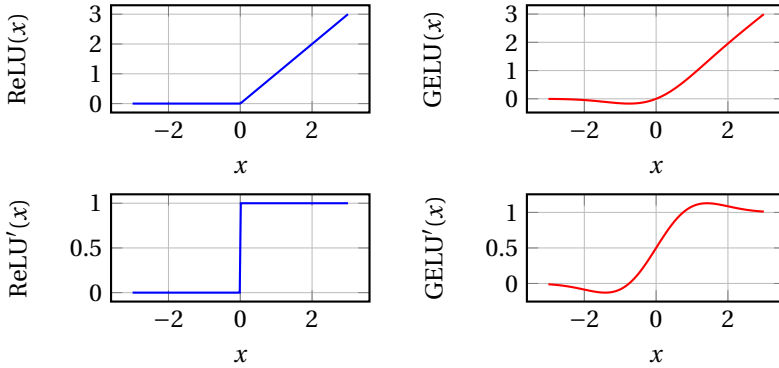


Figure 3.2: ReLU (left) and GELU (right) functions (top) and derivatives (bottom). ReLU does not allow gradients to propagate across negative activations due to its derivative there being equal to zero.

Activation Function	Precision Change
Baseline (GELU)	0.0 %
ReLU	-1.7 %
GRELU	+2.6 %

Table 3.1: Precision change in relation to baseline of activation functions. *GRELU* outperforms GELU and ReLU despite equal computational complexity to ReLU during inference.

3.2. GRELU

Our first optimization concerns the use of GELU. This activation function uses multiplication and one of either the error function, the hyperbolic tangent, or the sigmoid function, depending on the approximation used. These functions have a non-trivial hardware cost to implement [60, 7]. Therefore we aim to replace the GELU with a simpler alternative in order to improve hardware implementation cost.

We first investigate the use of the ReLU. This activation function has a very low computational cost [67], requiring only a single comparator and multiplexer in a hardware implementation. However, when used in the baseline in place of the GELU it causes an accuracy deficit of 1.7%. Therefore a solution is required to recover the lost accuracy.

Our contribution is to create a custom activation function, the *GRELU*, which uses the ReLU for the forward pass, and the GELU’s gradient for the backward pass. See Figure 3.2 for a visualization of the functions and their

Optimization	Precision Change
Baseline (GLU)	0.0 %
GLU with GRELU	+3.5 %
MLP	+3.5 %

Table 3.2: Precision impact in relation to baseline of alternatives to GLU. The MLP consists of a linear layer, GRELU activation function, and a second linear layer. The MLP has the same positive impact as GLU with GRELU despite a lower computational complexity.

derivatives. The *GRELU* is especially well-suited to our use-case, as training is performed off-chip and inference is performed on-chip. Therefore the hardware cost is identical to the ReLU. With this scheme, we recover the lost accuracy and further achieve an increase of 2.6%. See Table 3.1 for a summary.

A possible explanation of the *GRELU*'s performance may be related to the exact nature of the ReLU and inexact nature of the GELU. When evaluated on a floating point number, the GELU may introduce a finite computational error [22]. The ReLU on the other hand outputs either zero or the original input, thus always being exact. As we only consider the last output for the purposes of classification, hidden states must be updated 16,000 times, thus making the accuracy-preserving nature of the ReLU valuable. During the backwards pass, we do use the output at each time step to propagate the loss, thus making computational accuracy over time less important, while also benefiting from GELU's ability to propagate gradients through negative activations.

3.3. GLU Activation Function

The gated linear unit (GLU, see Section 2.2) as used in the baseline architecture includes a sigmoid function, giving it a high requirement in terms of hardware resources [60]. Therefore we focus on removing the need for a sigmoid function in these units. One possibility is to simply replace sigmoid by a less costly alternative such as the *GRELU*. Since the GLU is often used as a more expressive alternatives to the simpler multi-layer perceptron (MLP) [11], another alternative is to replace the GLU by an MLP. In Table 3.2 we show the impact of these alternatives. We find both achieve nearly the same accuracy increase, likely thanks in part to the use of the GRELU, and thus opt to use the simpler MLP.

	D	D = 0
C	470k, 0.0 %	414k, -5.57%
C = I	414k, -0.12%	286k, -1.2%

Table 3.3: Parameter count and precision change when dropping matrices from the LRU’s update and output functions. Dropping both only reduces the accuracy by -1.2% while reducing memory requirements to 60.7%.

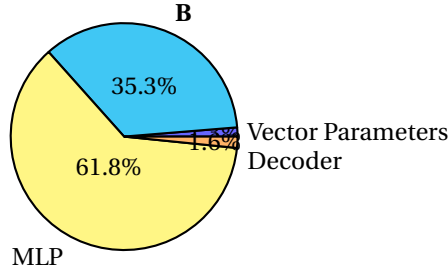


Figure 3.3: Parameter count per neural network component. Most are concentrated on dense layers, the MLP and **B**. Thus quantizing these will have the largest impact.

3.4. LRU Simplification

In order to remove unnecessary parameters, we experimented with removing matrix multiplications from the LRU’s output function. The idea is to simplify the output function as much as possible, restricting all computation within the LRU to the recurrent part. In Equation 2.5, **C** can be omitted by setting the output dimension equal to the hidden dimension. The **D** part of the equation can simply be omitted. In other words, **C** can be made equal to the identity matrix and **D** to the zero matrix. Table 3.3 shows the impact of these changes. Given the low accuracy drop relative to the parameter count change, we decide to remove both of these matrices.

The low impact of these omissions may be explained by the sequence and structure of the network. **D** provides a direct path between input and output, which may become less valuable given the long-range task and therefore higher reliance on the hidden state. Since the LRU is soon followed by a linear layer in the form of the MLP, this could be seen as assuming the role of **C**.

Matrix weight quantization	Precision
32-bit IEEE floating point (Baseline)	0.0 %
16-bit brain floating point	-0.30 %
16-bit IEEE floating point	+0.21 %
16-bit posit (E=2)	-4.47 %
8-bit posit (E=2)	-84.15 %
8-bit floating point (E=4, M=3)	+1.31 %

Table 3.4: Impact on network precision when applying quantization based on several number formats. Both the weight matrices and input activations are quantized, to simulate the usage of operations done at the quantized precision. See [29, 33] on the posit number format. E and M refer to exponent and mantissa size respectively. We choose 8-bit floating point due to its high performance and 4× size reduction compared to full precision.

3.5. Matrix Multiplication Quantization

Our final optimization concerns quantization of operations in the neural network. Most parameter count is concentrated on the fully connected layers and LRU matrix **B**, as seen in Figure 3.3. Therefore, minimizing weight matrix parameter bit count reduces the memory impact of 97.1% of the parameters. Additionally, we find a factor of 100× more operations originating from matrix multiplication compared to vector operations in the modified architecture. This means a reduced processing rate for vector operations is acceptable, thus permitting a greater amount of hardware resources to be used per vector operation even at the cost of throughput.

Based on these observations, we resolve to make use of quantization only for network weights and inputs to matrix-vector multiplications. Inputs are quantized as otherwise full size multipliers would be required, nullifying the hardware resource count advantage of quantization. Outputs of matrix-vector multiplication need not be quantized as it is possible to accumulate to a larger bit width and convert to full precision outputs before passing results to the vector unit. These quantized parts of the network are underlined in Figure 3.1.

We test this quantization scheme on various number formats. Table 3.4 shows the results of our experiments. Based on this, We choose 8-bit floating point with a 4 bit exponent and 3 bit mantissa. This number format has the advantage of maintaining a good level of precision. At the same time, its mantissa can be multiplied using a 6-input lookup table (LUT), as discussed later in Section 5.3.

4

Hardware Architecture

An FPGA-based accelerator was developed based on the optimizations discussed in the previous chapter, and the computations required as illustrated by Figure 4.1. The accelerator is for the most part general purpose, able to flexibly implement models based around fully-connected layers. However, the positive results of the quantization experiments specific to the LRU model in the previous section justify a mixed-precision implementation between matrix and vector operations. Additionally, the vector unit is extended to implement all of the LRU's vector operations. This chapter discusses the architecture of the various components of the accelerator, including the LRU-specific design decisions.

4.1. Overview

At a high level, the accelerator is divided into a control unit, vector memory, matrix unit and vector unit. Figure 4.2 illustrates the different components of the accelerator. Dedicated vector and matrix parameter memory is used in order to allow for concurrent accesses, see Brainwave for a similar concept [15].

The control unit contains the registers from Table 4.1. When a command is dispatched by setting the corresponding register bit, the very long instruction word (VLIW) command is split into several smaller commands, which are then queued by the different parts of the accelerator. This allows for up to 16 commands to be queued at once, saving CPU waiting time, and allows for high frequency by registering the path from the control unit to the other

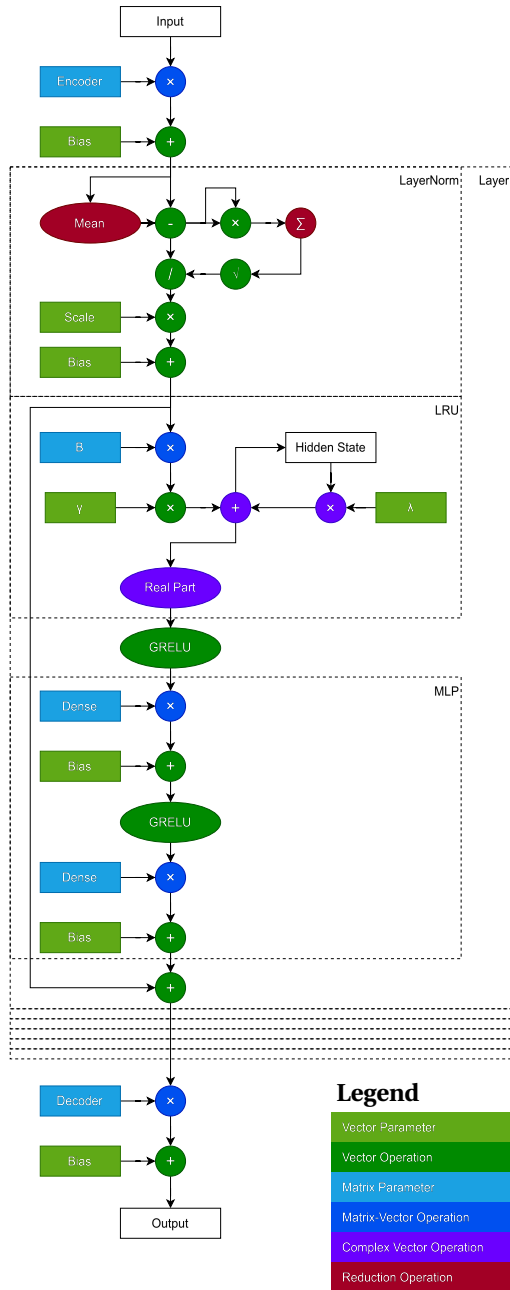


Figure 4.1: Diagram showing computations as performed by our simplified LRU architecture, representing the operations the accelerator must implement and their order. Note the six stacked layers.

AddressOffset	RegName	Description	Width	Section	FieldName	R/W	Reset value	Field-Description
0x0000	STATUS	Status	32	[31:2]	–	NA	30'b0	Reserved
				[1]	storeIdle	RO	1'b0	Store unit idle
				[0]	ready	RO	1'b0	Command FIFO not full
0x0004	CONTROL	Control	32	[31:28]	–	NA	4'b0	Reserved
				[27:25]	vecRhs	RW	3'b0	RHS for vector unit (0 = parameter, 1 = duplicate lhs, 2 = reduction register, 3 = previously loaded vec value, 4 = constant register, 5 = complex identity)
				[24:20]	blockCount	RW	5'b0	How many vector blocks or MAC rows to load
				[19:12]	lhsAddr	RW	8'b0	LHS address (Vector memory)
				[11:1]	rhsAddr	RW	11'b0	Vector unit RHS address (Vector mem or parameter address)
				[0]	dispatch	W1P	1'b0	Send command to FIFO
0x0008	MAC	Matrix unit control	32	[31:14]	–	NA	18'b0	Reserved
				[13:9]	macCols	RW	5'b0	How many MAC columns to load
				[8:0]	macParamAddr	RW	9'b0	Address of MAC blocks to start loading
0x000C	CONTROL2	Control	32	[31:14]	–	NA	18'b0	Reserved
				[13]	useAccumRsqr	RW	1'b0	Use reciprocal square root upon finalizing accumulation
				[12:10]	vecMode	RW	3'b0	Vector computation. (000 = add, 001 = mul, 010 = accum, 011 = complex real part, 100 = complex imaginary part, 101 = complex full)
				[9]	vecLhs	RW	1'b0	LHS for vector unit (0 = mac, 1 = vec)
				[8]	storeRelu	RW	1'b0	Apply RELU before storing results
				[7:0]	storeAddr	RW	8'b0	Address where to store results
0x0010	CREG	Constant register	32	[31:0]	creg	RW	32'b0	
0x0014	RREG	Reduction register	32	[31:0]	rreg	RO	32'b0	

Table 4.1: Registers exposed by accelerator. These are used by the host CPU to issue commands.

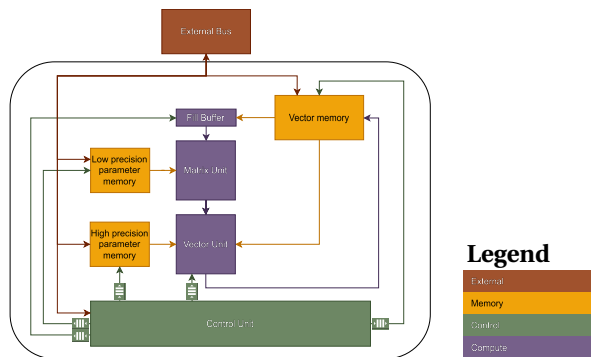


Figure 4.2: High level architecture of the accelerator. Unlabeled rectangles represent FIFOs. The black outline represents the boundary of the accelerator. Our design allows for acceleration of LRU networks.

components. The inability to register such paths may have halved the design's maximum frequency.

This control paradigm was chosen mainly due to its ease of implementation and use, as it requires no additional finite-state machines (FSMs) at the control unit level to send commands and is straightforward to program for. However, another consideration was to ease the transition to a more efficient control paradigm, such as the use of an instruction buffer. Since all units receive commands independently, only the control unit would need to be modified. This would decrease the need for CPU to accelerator communication on the order of $1000\times$. However, it would necessitate additional synchronization features such as waiting on the completion of a previous operation before starting the next. In our design, such a change would only represent the addition of several dozens of LUTs.

Care is taken to ensure each command corresponds to a complete vector or matrix-vector operation, easing the bandwidth required between CPU and accelerator. Each of these commands begins with a source vector and targets a destination vector in vector memory. The exception are vector accumulation operations whose target is the reduction register (RREG). This further simplifies the programming model.

4.2. Vector Unit

The vector unit was designed to implement all vector computations in our simplified LRU architecture. See Figure 4.3 for an illustration. Due to the

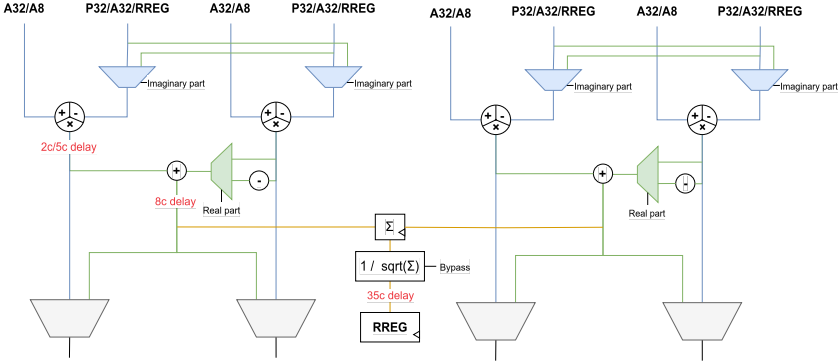


Figure 4.3: Four-input example of reconfigurable vector unit. Red labels show cycle delay for each type of operation. "Imaginary part" and "Real part" inputs are used as inputs for complex number multiplication and are both set to false in the case of real numbers. The datapath for simple operations is denoted in blue, for complex operations in green, and for accumulation in orange. The vector unit implements all vector operations used in the LRU.

variety in required operations, and the non-overlapping nature of their execution, our design efforts focused on finding a suitable architecture with as many shared components as possible. This led us to a layered architecture, where successive layers are used to implement different operations, utilizing the resources of layers below. The following operations are implemented:

- *Simple operations.* This covers element-wise multiplication and addition. The vector unit uses only the first level of multipliers or adders.
- *Complex multiplication.* In order to perform complex multiplication, the vector unit uses the first level multipliers together with the second level adders. Depending on the option selected, the vector unit produces either the real or imaginary part of the result.
- *Summation.* This is used in order to sum a vector, mainly for the implementation of normalization operations as in LayerNorm [1]. It makes use of either the first level multipliers or adders, the second stage adders, and an adder tree leading to a floating point accumulator. The output of the accumulator includes an optional reciprocal square root stage, useful for computing reciprocal standard deviation. When accumulation is complete, the reduction result can be used as an input to the vector unit, denominated reduction register "RREG".

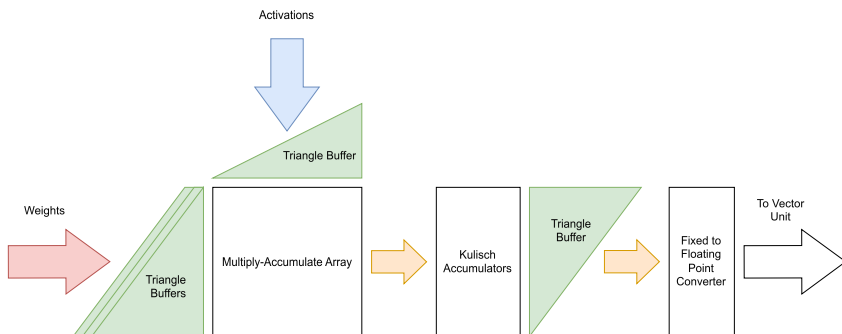


Figure 4.4: Matrix unit. Vector inputs flow from top to bottom. Partial sums from left to right. Delay registers are added in order to correct weight and vector input arrival timing. This design was chosen to optimize for register-to-register delay while reaching a reasonable compromise on latency.

Given the variability in latency of the different types of operations, one key problem to solve was conflicts between operations and orderedness of results. For example, if an operation with a longer latency would be issued right before an operation with lower latency, then either their outputs would conflict or the lower latency operation would output its result first, i.e. out of order. We solve this by employing a set of counters to generate backpressure. These counters are set every time an operation is accepted and ensure no operation can enter whose latency is lower than the remaining cycles for the last operation. In this way, full throughput is preserved for operations with the same latency, and results of the vector unit are always in order.

Some additional minor components are required to support complex multiplication. Since the vector unit produces results at half-rate for complex multiplication and requires input values to be available for two cycles, a fill buffer is used in the output path of the vector unit in order to gather complex results. Another unit in the input path repeats inputs twice in the case of complex operations.

4.3. Matrix Unit

The matrix unit is used in order to accelerate matrix-vector multiplication operations quantized according to Section 3.5 and is shown in Figure 4.4. One key concern is to allow for high operating frequency despite the long worst-case critical path between input and result in a regular multiply-accumulate array. Therefore we make use of a systolic architecture. However,

since our implementation performs unbatched matrix-vector multiplication, then weight reuse and therefore weight stationarity cannot be exploited [56]. Regardless, register-to-register paths are shortened compared to a combinatorial design. Additionally, since input sources to each PE are fixed, block rams are positioned near their corresponding PEs. This greatly reduces the amount and length of wiring to matrix parameter memory.

An important point mentioned in Section 3.5 is the ability to pass full precision results to the vector unit. This implies the results of matrix-vector multiplication should be accumulated into registers of higher precision. We identify the following solutions:

- Floating-point accumulator, where the partial sums would be accumulated into the higher precision required by the vector unit. This necessitates the use of a floating-point adder, which for 32-bit floating point requires on the order of hundreds of LUTs and pipelining if frequency is to be on the hundreds of MHz. The pipelining presents a further issue as it implies a delay between output and result but in accumulation every input depends on the previous result. This can be solved by way of stalling, reducing throughput. Alternatively, several independent sums can be allowed to be in flight and reduced after the last input is received [2], adding latency and resulting in an unpredictable result due to floating point's non-associativity;
- Kulisch accumulator [38], where partial sums are first translated into fixed point and then accumulated. Due to the use of fixed point, this type of accumulator is always associative and exact. However, for a 32-bit floating point it requires the use of a 554 bit fixed point number, therefore in some cases requiring complex schemes in order to achieve acceptable frequency [61].

We choose the Kulisch accumulator due to its greater resource efficiency. Our partial sums result from the product of two 8 bit floating-point numbers, each with an exponent range of 2^7 to 2^{-10} , and 3 mantissa bits. Since multiplication involves the addition of the exponents and multiplication of a mantissa, we can fit the result of multiplying two of these numbers exactly in an exponent range of 2^{14} to 2^{-20} and 6 mantissa bits. This gives us a fixed-point number size of 40 bits, able to represent any such product exactly. An adder for such a fixed-point number can be formed from cascading five 8-bit adders in our chosen FPGA architecture. This represents a lower requirement compared to the hundreds of LUTs needed for a floating point accumulator. Given the relatively small size of this exact representation, we also use it for

the partial sums. Therefore, the results of our matrix-multiplication block are always exact given 8-bit floating point inputs.

In order to allow for greater flexibility in the design space of the accelerator, we support differing input and output sizes for the matrix unit. However, vector memory must provide inputs to both the matrix and vector units, and is therefore sized according to the output size of the matrix unit, which is the input size of the vector unit. Therefore, we solve this issue by placing a fill buffer in front of the matrix unit. Values from the vector memory are converted into 8-bit floating point numbers and read into the fill buffer until an entire vector has been processed. Then this vector is read from the fill buffer several times to complete the matrix-vector operation and vector memory does not need to be active during this time.

5

Results

5.1. Experimental Setup

We implement an end-to-end accelerator in SpinalHDL and evaluate it using our modified LRU model (See Chapter 3). We perform classification on the Google Speech Commands dataset [64], including all 35 labels and operating on 16 KHz sound files. Our design is synthesized using Vivado 2023.2¹ and targets a Xilinx Zynq ZCU104 development board at 350 MHz. Our design resides in the programmable logic (PL) and is sent commands from the on-chip Cortex A53 CPU via memory-mapped IO using the Pynq² framework. All network weights are stored directly in the PL.

In order to validate the design, we store layer-by-layer outputs of the network as computed by the JAX³ implementation. Then, the same computations are performed on the PL and check for an exact match. Our flow is illustrated in Figure 5.1.

Important Note on Power and Energy Statistics

Power estimation was performed without the use of signal switching statistics due a lack of support for SAIF export when using Vivado's XSI for simulation. Therefore power and energy statistics should only be taken as an indication of actual accelerator characteristics.

¹<https://www.xilinx.com/products/design-tools/vivado.html>

²<https://www.pynq.io/>

³<https://github.com/google/jax>

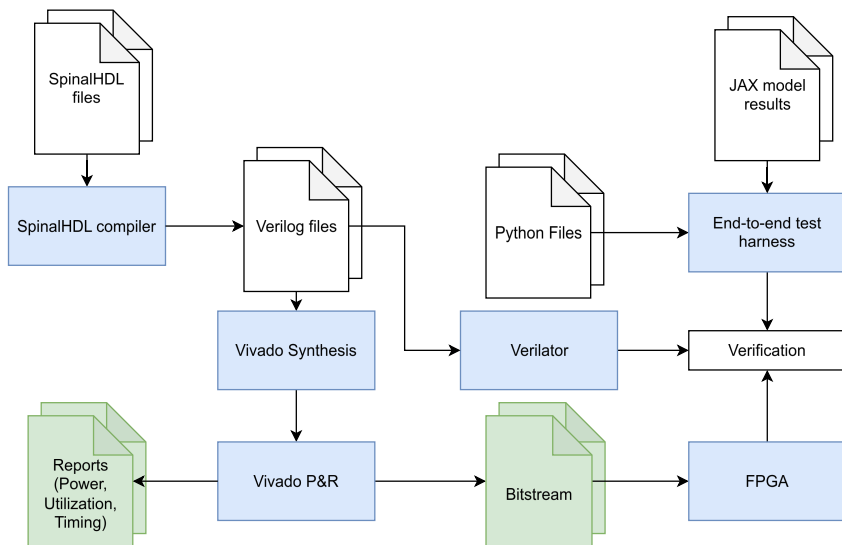


Figure 5.1: Tool flow for validation and experiments. SpinalHDL is the hardware definition language used to define the accelerator and its components. It is then compiled to Verilog, a more primitive hardware definition language parsable by the Vivado synthesizer. The synthesis step maps the Verilog code to the primitives provided by the FPGA technology. Place and route (P&R) maps these elements onto physical resources on the FPGA device. This can then be used to create a bitstream, the proprietary binary file used by the FPGA to determine the configuration of its components and the connections between them. Once the design is loaded on the FPGA, we validate the design by running the model end-to-end on the FPGA and comparing the output for each layer to the results of a reference JAX implementation. Beforehand, we also perform the same process via behavioural simulation with Verilator, an open source Verilog simulator.

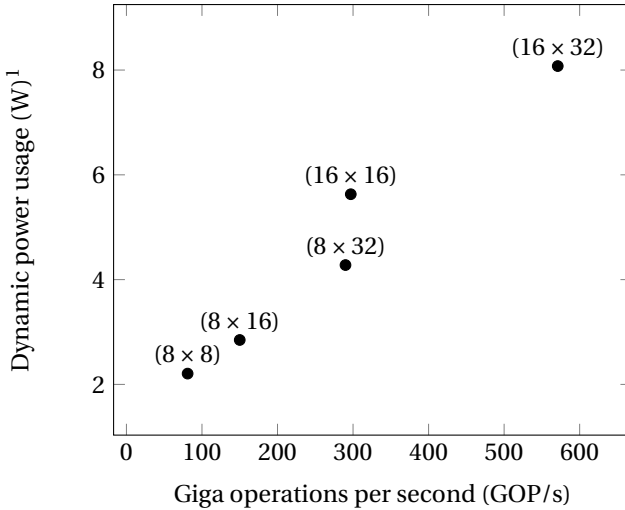


Figure 5.2: Scatter plot comparing operations per second with power usage for various configurations of the form (rows,columns). Despite resulting in the same amount of matrix PE's, a jump between (8,32) and (16,16) takes place due to the vector unit's size being determined by the number of rows. A vector PE operates on 32 bit floating-point numbers versus the matrix unit's PE of 8 bits, thereby using more resources. Based on these results we choose 8 rows and 32 columns. ¹Lacks activity annotation.

5.2. Design Space Exploration

Two important variables in the performance of our accelerator are the dimensions of the matrix and vector unit. Therefore, we perform a design space exploration in order to find the optimal configuration in terms of GOP/s/W, while still running our task in real time.

Figure 5.2 shows the result of our exploration. We choose 8 rows and 32 columns, as this configuration attains the minimum power usage while still allowing for real-time inference on our task.

5.3. Resource Utilization

Table 5.1 shows the overall resource use of the accelerator, with Figure 5.4 showing the corresponding device view and Figure 5.3 breaking down BRAM usage by component. Note the usage of only 16 DSPs. This is due to the use of 8-bit floating-point numbers. With a 3-bit mantissa, only a 3-bit multiplier is needed for floating-point multiplication. This allows us to use a 6-input

Resource	Amount	Percentage
CLB LUTs	62285	27.03%
CLB Registers	40805	8.86%
CARRY8	1975	6.86%
F7 Muxes	1108	0.96%
F8 Muxes	401	0.96%
F9 Muxes	0	0.00%
18K BRAM	352	52.08%
URAM	0	0.0%
DSP	16	0.93%

Table 5.1: Resource utilization of our accelerator on ZCU104 FPGA. The accelerator has a low resource requirement compared to the total available.

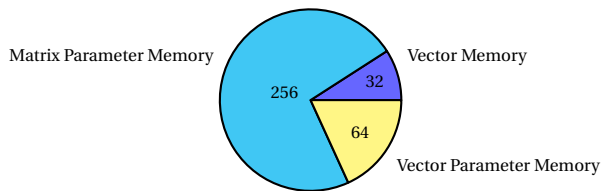


Figure 5.3: BRAM use broken down by component. Despite being quantized, most memory use is concentrated in matrix parameters, used for fully-connected layers.

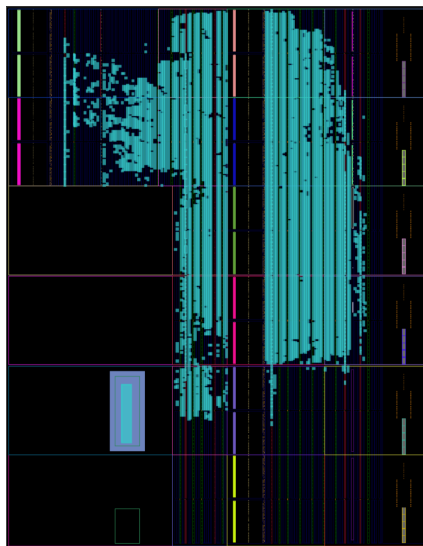


Figure 5.4: Device view highlighting resources used as shown by Vivado. The FPGA resources in use by the design are colored light blue, with the unused elements colored black. The design uses 27.03% of LUTs and 52.08% of BRAM.

Operation	Latency	Throughput
Matrix-vector multiplication	41	8
Complex parameter-vector multiplication	7	4
Vector-parameter multiplication	2	8
Vector-parameter addition	5	8
Vector-vector addition	5	4
Vector summation	35	N/A

Table 5.2: Latency and throughput of various operations. Latency is given in cycles and throughput in output elements per cycle. Low latency helps to minimize stalling when performing back-to-back dependent calculations.

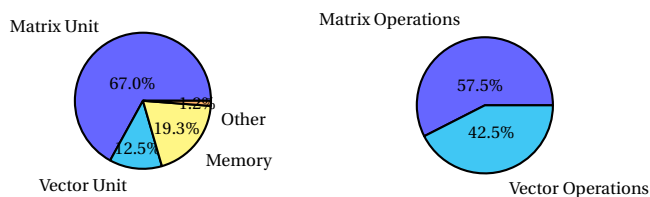


Figure 5.5: Proportion of power usage for each component of the accelerator (left) and proportion of cycles spent on each type of operation (right). The matrix unit and its operations dominate in both categories. Note that power usage was calculated without activity annotation.

lookup table in its place. Therefore, the DSPs are only used in the vector unit, with two DSPs needed per 32-bit floating-point number.

5.4. Hardware Architecture Characteristics

Latency is made an important concern due to our objective of real-time inference. Excessive latency is undesirable as it may necessitate the use of barriers to wait for previous results. Latency and throughput numbers are given in Figure 5.2. In the case of vector summation the dimension of the result is always one, and if the next operation requires the result (which is the case in our implementation), then a pipeline stall of 35 cycles is incurred as the accumulator finishes its work.

5.4.1. Power Usage by Component

Figure 5.5 shows the relative power usage of each component in the accelerator and compares it to the amount of cycles taken up by each type of operation. The matrix unit has a slightly disproportional power usage to cycle

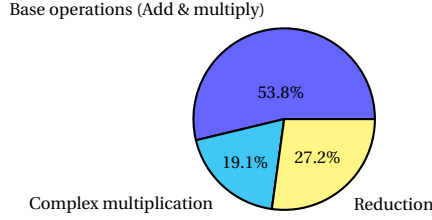


Figure 5.6: LUT utilization ratio of base and special functions of the vector unit. Functionality additional to the base operations has a lower resource requirement thanks to hardware reuse.

use ratio. Nevertheless, this is necessary in order to achieve real time inference. In order to address this, the matrix unit's columns could be enlarged and its rows (corresponding to the vector unit's size) could be made smaller. This would close the gap, but at the cost of higher power usage, and no benefit towards satisfying the accelerator's throughput requirements (See Figure 5.2).

5.4.2. Special Function Overhead

Figure 5.6 compares the hardware overhead of complex multiplication and reduction with respect to the vector unit's base operations. Complex multiplication requires the use of one additional adder per two elements. Reduction makes use of the hardware for addition and complex multiplication, and in addition requires an adder tree. Therefore, the hardware use for reduction has a logarithmic dependency on vector unit size and would be lessened in wider vector units.

5.4.3. Approach to LayerNorm

Since the use of LayerNorm is common in transformers-based architectures [13, 52, 8], hardware acceleration approaches have been developed for it. SwiftTron [46] addresses the issue by generating one LayerNorm unit per vector output. Inputs into the LayerNorm unit are directed into an accumulator, whose output can optionally be processed by a square root unit. The main differences with our approach is that this is a separate unit not reusing hardware components. Additionally, since we target real-time inference, we are unable to use a LayerNorm unit per output, as the presence of batching makes possible in SwiftTron.

In [44], an approach to layer normalization is proposed where the reduction for LayerNorm is computed at the same time as the corresponding out-

put becomes available. Additionally, variance is reformulated as to not depend on the mean. The first part of this approach is compatible with our accelerator, as accumulation can be performed as an output becomes available. The second part is also possible, but would require double the amount of adders due to two reductions being calculated at once.

5.4.4. Hardware-friendly Activation Function

In this work we proposed the *GRELU*, a GELU-ReLU activation function hybrid using ReLU in the forward pass and GELU in the backward pass. [31] proposes two new activation functions with simple implementations in hardware and comparable performance to GELU: DNR and PWL. Our approach is more efficient during inference, due to the extreme simplicity of the ReLU activation function. However, if on-chip training is desired, the gradients of DNR and PWL are more easily implemented in hardware.

5.5. Comparison with other works

In this section, we compare our accelerator to other works. Most existing keyword-spotting accelerators target ASIC platforms and use networks with different computational requirements compared to the LRU. Therefore, we split our comparison between accelerators which are implemented in a similar technology and mainly accelerate fully-connected layers, and accelerators targeting the Google Speech Commands Dataset.

Table 5.3 compares our accelerator to others with computationally similar workloads. We achieve competitive GOP/s/W, achieving $2.64\times$ the runner-up’s performance though lacking activity annotation in our case. This is mostly due to our use of 8-bit floating point which does not require use of limited DSPs, as discussed in Section 3.5. Another contributing factor is our accelerator’s maximum frequency, which at $1.4\times$ the runner-up’s performance allows higher throughput with the same amount of hardware components. However, this in turn contributes to our accelerator using the highest amount of power, further exacerbated by the inclusion of vector operations, as discussed previously in Section 5.4.1. This is at $1.2\times$ compared to another floating point design, and $204\times$ compared to a fixed point design.

Table 5.4 compares our accelerator to other designs implementing keyword spotting on the Google Speech Commands dataset. We achieve the highest classification accuracy, at a 0.2% lead compared to runner-up [30]. However, this comes at a cost, as [30] only uses 20% of the memory and 0.8% of the power. This can be explained by differences in the two network architectures. While we operate on the full 16,000 samples and must apply our

	2022 [30]	2023 [59]	Ours
Platform	Zynq 7Z020	Zynq ZCU104	Zynq ZCU104
Maximum Frequency (Mhz)	50	200	350
Numeric Precision	8-bit fixed point	16-bit brain floating point	32-bit/8-bit floating point
Memory (KB)	100	990	488
Power usage (mW)	209	3364	[4278] ^{1,2}
Efficiency (GOP/s/W)	25.6	15.895	≈ 70 ^{1,2}

Table 5.3: Accelerator compared to other works also mainly accelerating fully-connected layers and implemented on a similar technology. Our design demonstrates best in class efficiency and maximum frequency, with competitive power usage to other floating point accelerators.

¹Calculated at the maximum frequency. ²Lacks activity annotation.

network weights on each sample, [30] uses MFCC to reduce the number of samples to 62 and due to the convolutional nature of TENet, only applies its weights once. On the other hand, this larger amount of total computations in our accelerator means the time to process the last sample and thus final output latency is only $5\mu s$, 0.04% of the runner-up [32]. The only other accelerator not using MFCC is [69]. However, this accelerator also uses a convolutional architecture thus being able to compute some of the intermediate activations ahead of time but still requiring the last sample in order to compute much of the tree. Note also that our accelerator classifies the dataset's full 35 keywords compared to the rest of the works compared only classifying 10, and accuracy may change depending on the number of labels used.

	2020 [5]	2020 [21]	2022 [30]	2023 [69]	2023 [32]	Ours
Network Structure	TC-ResNet8	LSTM	TENet6	BTCN	CTN	LRU
Platform	22nm (ASIC)	65nm (ASIC)	Zynq 7Z020 (FPGA)	Cyclone V (FPGA)	22FDX (ASIC)	Zynq ZCU104 (FPGA)
Numeric Precision	6/8-bit fixed point	4-bit LUT/8 bit fixed point	8-bit fixed point	-	8-bit fixed point	32/8-bit floating point
Minimum Real-Time Frequency (Mhz)	0.25	0.25	0.09	-	-	200
Memory (KB)	76	65	100	56	132	488
Accuracy (%)	93.09	90.87	95.36	90.3	93.3	95.50
Keywords	10	10	10	10	10	35
Latency (ms)	100	15	100	19.3	11	0.005
Power Usage (mW)	0.0082	0.0106	209	156.9	0.193	[2444.6] ^{1,2}
MFCC	Yes	Yes	Yes	No	Yes	No

Table 5.4: Accelerator compared to other works implementing keyword spotting on GSCD. Information not available is denoted with "-". Latency denotes the time between the last sample being received by the accelerator and a classification output being emitted. Our design achieves best in class accuracy, latency and keyword amount. ¹Calculated at the minimum real-time frequency. ²Lacks activity annotation.

6

Conclusion

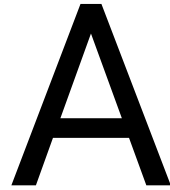
In this work we aim to explore possible avenues for LRU hardware-software co-design in order to allow for power-efficient and capable sequence modeling. To this end, we present an LRU accelerator focusing on power efficiency when performing keyword spotting on raw audio data. We perform several optimizations to the model including the use of an MLP instead of a GLU, the removal of low-impact components in the LRU’s output and hidden state functions, quantization of weight matrices, and the use of our novel *GRELU* activation function. Our hardware accelerator is able to perform all operations required for our modified LRU network, efficiently implementing complex multiplication and LayerNorm using a single novel vector unit architecture. This first LRU accelerator paves the way for future work centered around LRU hardware acceleration.

Compared to other similar accelerators, we achieve state-of-the-art accuracy, latency, and maximum frequency, at 95.5% @ 35 keywords, 5 μ s, and 350 MHz respectively with comparable efficiency at ≈ 70 GOP/s/W. However, other metrics lag behind other accelerators such as power consumption and memory requirements, at 488 KB and 2.44 W respectively. That being said, possible avenues for future work include:

- *on-chip learning* using an online learning rule such as in [76], which would extend the usefulness of the accelerator by allowing it to fine-tune on a speaker or learn new keywords,
- *block floating point formats*, such as proposed in [54], in order to fur-

ther reduce weight size and computation complexity, with a further 2x-4x memory use reduction possible,

- *temporal and spatial sparsity*, such as proposed in [43, 12] in order to reduce the operation count requirement for real time inference by up to 100x and thereby improve energy efficiency,
- *application on datasets with larger sample size*, such as [45, 68, 49], as these could benefit from the LRU's suitability to longer-scale dependencies, and thus extend an LRU accelerator's edge over those implementing other model architectures.



Model Training Parameters

Name	Value
Hidden Dimensions	128
Model Dimensions	96
Batch Size	16
Epoch	100
Learning rate	2e-3
Learning factor	3
Layers	6
Normalization Type	LayerNorm
Optimizer	Adam
Dropout probability	0.1%
Dense Layer Quantization Format	float8_e4m3fnuz
LRU λ minimum	0.99
Weight decay	4%
Learning Rate Scheduler	Cosine Annealing
LRU Components	B , λ
Activation Function	GReLU

Bibliography

- [1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. arXiv:1607.06450 [cs, stat]. July 2016. DOI: 10.48550/arXiv.1607.06450. URL: <http://arxiv.org/abs/1607.06450> (visited on 07/10/2024).
- [2] Marco Bassoli, Valentina Bianchi, and Ilaria De Munari. “A Simulink Model-Based Design of a Floating-Point Pipelined Accumulator with HDL Coder Compatibility for FPGA Implementation”. en. In: *Applications in Electronics Pervading Industry, Environment and Society*. Ed. by Sergio Saponara and Alessandro De Gloria. Cham: Springer International Publishing, 2020, pp. 163–171. ISBN: 978-3-030-37277-4. DOI: 10.1007/978-3-030-37277-4_19.
- [3] Y. Bengio, P. Simard, and P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994). Conference Name: IEEE Transactions on Neural Networks, pp. 157–166. ISSN: 1941-0093. DOI: 10.1109/72.279181. URL: <https://ieeexplore.ieee.org/document/279181> (visited on 07/22/2024).
- [4] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation”. In: *ArXiv* (Aug. 2013). URL: <https://www.semanticscholar.org/paper/Estimating-or-Propagating-Gradients-Through-Neurons-Bengio-L%C3%A9onard/62c76ca0b2790c34e85ba1cce09d47be317c7235> (visited on 07/06/2024).
- [5] Paul Palomero Bernardo et al. “UltraTrail: A Configurable Ultralow-Power TC-ResNet AI Accelerator for Efficient Keyword Spotting”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (Nov. 2020). Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 4240–4251. ISSN: 1937-4151. DOI: 10.1109/TCAD.2020.3012320. URL: <https://ieeexplore.ieee.org/abstract/document/9216480> (visited on 08/05/2024).

- [6] Guy E. Blelloch. “Prefix sums and their applications”. In: (1990). Publisher: School of Computer Science, Carnegie Mellon University Pittsburgh, PA, USA. URL: <http://shelf2.library.cmu.edu/Tech/23445461.pdf> (visited on 07/03/2024).
- [7] Safa Bouguezzi, Hassene Faiedh, and Chokri Souani. “Hardware Implementation of Tanh Exponential Activation Function using FPGA”. In: *2021 18th International Multi-Conference on Systems, Signals & Devices (SSD)*. ISSN: 2474-0446. Mar. 2021, pp. 1020–1025. DOI: 10.1109/SSD52085.2021.9429506. URL: <https://ieeexplore.ieee.org/document/9429506> (visited on 07/29/2024).
- [8] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. July 2020. DOI: 10.48550/arXiv.2005.14165. URL: <http://arxiv.org/abs/2005.14165> (visited on 07/13/2024).
- [9] Chixiao Chen et al. “OCEAN: An on-chip incremental-learning enhanced processor with gated recurrent neural network accelerators”. In: *ESSCIRC 2017 - 43rd IEEE European Solid State Circuits Conference*. Sept. 2017, pp. 259–262. DOI: 10.1109/ESSCIRC.2017.8094575. URL: https://ieeexplore.ieee.org/abstract/document/8094575?casa_token=m2HazcNZiNIAAAAA:gwqZ0isAh7-J7C_b6sUWeT-GwOk4wVvXf-AqHrsd2s36gLyWD20m5ZpULFw9A7jz5qXeuHU (visited on 09/25/2023).
- [10] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. *A Computational Model for Tensor Core Units*. arXiv:1908.06649 [cs]. July 2020. DOI: 10.48550/arXiv.1908.06649. URL: <http://arxiv.org/abs/1908.06649> (visited on 04/01/2024).
- [11] Yann N. Dauphin et al. “Language modeling with gated convolutional networks”. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70. ICML’17*. Sydney, NSW, Australia: JMLR.org, Aug. 2017, pp. 933–941. (Visited on 07/23/2024).
- [12] Tobi Delbruck and Shih-Chii Liu. “Data-Driven Neuromorphic DRAM-based CNN and RNN Accelerators”. In: *2019 53rd Asilomar Conference on Signals, Systems, and Computers*. ISSN: 2576-2303. Nov. 2019, pp. 500–506. DOI: 10.1109/IEEECONF44664.2019.9048865. URL: https://ieeexplore.ieee.org/abstract/document/9048865?casa_token=NQUSKR7b4EcAAAAA:jd3kQnYju6HtKo2o81Bu7JmmkbcI1clK89KDA07QWOHLVpnAaFEz-GH9wew7SIObmHKs9UY (visited on 10/02/2023).

- [13] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv:1810.04805 [cs]. May 2019. DOI: 10.48550/arXiv.1810.04805. URL: <http://arxiv.org/abs/1810.04805> (visited on 07/13/2024).
- [14] Jeffrey L. Elman. "Finding Structure in Time". en. In: *Cognitive Science* 14.2 (1990). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1207/s15516709cog1402_1, pp. 179–211. ISSN: 1551-6709. DOI: 10.1207/s15516709cog1402_1. URL: https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog1402_1 (visited on 10/18/2023).
- [15] Jeremy Fowers et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. ISSN: 2575-713X. June 2018, pp. 1–14. DOI: 10.1109/ISCA.2018.00012. URL: <https://ieeexplore.ieee.org/abstract/document/8416814> (visited on 07/10/2024).
- [16] Charlotte Frenkel and Giacomo Indiveri. "ReckOn: A 28nm Sub-mm2 Task-Agnostic Spiking Recurrent Neural Network Processor Enabling On-Chip Learning over Second-Long Timescales". In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 65. ISSN: 2376-8606. Feb. 2022, pp. 1–3. DOI: 10.1109/ISSCC42614.2022.9731734. URL: <https://ieeexplore.ieee.org/abstract/document/9731734> (visited on 10/06/2023).
- [17] Chang Gao, Tobi Delbruck, and Shih-Chii Liu. "Spartus: A 9.4 TOP/s FPGA-Based LSTM Accelerator Exploiting Spatio-Temporal Sparsity". In: *IEEE Transactions on Neural Networks and Learning Systems* (2022). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 1–15. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2022.3180209. URL: https://ieeexplore.ieee.org/abstract/document/9794448?casa_token=JrtX9479dHoAAAAA:qQqXUJHmOCSe0cR0ertJtGzsJT9CvH27ByZiJXrE4ToaA8350_kLK-FGS9hOmk7Td9foexs (visited on 10/02/2023).
- [18] Alireza Ghaffari et al. "Is Integer Arithmetic Enough for Deep Learning Training?" In: (2022). Publisher: arXiv Version Number: 3. DOI: 10.48550/ARXIV.2207.08822. URL: <https://arxiv.org/abs/2207.08822> (visited on 12/07/2023).

- [19] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. arXiv:2103.13630 [cs]. June 2021. DOI: 10 . 48550/arXiv . 2103 . 13630. URL: <http://arxiv.org/abs/2103.13630> (visited on 07/06/2024).
- [20] J.S.P Giraldo and Marian Verhelst. “Laika: A 5uW Programmable LSTM Accelerator for Always-on Keyword Spotting in 65nm CMOS”. In: *ESSCIRC 2018 - IEEE 44th European Solid State Circuits Conference (ESSCIRC)*. ISSN: 1930-8833. Sept. 2018, pp. 166–169. DOI: 10 . 1109/ESSCIRC . 2018 . 8494342. URL: https://ieeexplore.ieee.org/abstract/document/8494342?casa_token=vMcd71Z1BJcAAAAA:OU8nhxNSIJdRnrLJzn4qeNhxloXHKF4RY_jIuvn2bk2lwmNoqVYPsXP_Lri4awq0HnYjxdM (visited on 11/04/2023).
- [21] Juan Sebastian P. Giraldo et al. “Vocell: A 65-nm Speech-Triggered Wake-Up SoC for 10- u W Keyword Spotting and Speaker Verification”. In: *IEEE Journal of Solid-State Circuits* 55.4 (Apr. 2020). Conference Name: IEEE Journal of Solid-State Circuits, pp. 868–878. ISSN: 1558-173X. DOI: 10 . 1109/JSSC . 2020 . 2968800. URL: <https://ieeexplore.ieee.org/abstract/document/8978574> (visited on 08/05/2024).
- [22] David Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM Comput. Surv.* 23.1 (Mar. 1991), pp. 5–48. ISSN: 0360-0300. DOI: 10 . 1145 / 103162 . 103163. URL: <https://doi.org/10.1145/103162.103163> (visited on 07/29/2024).
- [23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. en. Google-Books-ID: omivDQAAQBAJ. MIT Press, Nov. 2016. ISBN: 978-0-262-33737-3.
- [24] Albert Gu and Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. arXiv:2312.00752 [cs]. Dec. 2023. DOI: 10 . 48550/arXiv . 2312 . 00752. URL: <http://arxiv.org/abs/2312.00752> (visited on 12/10/2023).
- [25] Albert Gu, Karan Goel, and Christopher Ré. *Efficiently Modeling Long Sequences with Structured State Spaces*. arXiv:2111.00396 [cs]. Aug. 2022. URL: <http://arxiv.org/abs/2111.00396> (visited on 07/03/2024).
- [26] Albert Gu et al. “HiPPO: Recurrent Memory with Optimal Polynomial Projections”. en. In: (2020).

- [27] Yunhui Guo. “A Survey on Methods and Theories of Quantized Neural Networks”. In: *ArXiv* (Aug. 2018). URL: <https://www.semanticscholar.org/paper/A-Survey-on-Methods-and-Theories-of-Quantized-Guo/6e5f2c7d1092d733ae85c269bf12e47755b90368> (visited on 07/06/2024).
- [28] Suyog Gupta et al. *Deep Learning with Limited Numerical Precision*. arXiv:1502.02551 [cs, stat]. Feb. 2015. DOI: 10.48550/arXiv.1502.02551. URL: <http://arxiv.org/abs/1502.02551> (visited on 07/06/2024).
- [29] Gustafson and Yonemoto. “Beating Floating Point at its Own Game: Posit Arithmetic”. In: *Supercomputing Frontiers and Innovations: an International Journal* 4.2 (June 2017), pp. 71–86. ISSN: 2409-6008. DOI: 10.14529/jsfi170206. URL: <https://doi.org/10.14529/jsfi170206> (visited on 02/24/2024).
- [30] Keyan He, Dihua Chen, and Tao Su. “A Configurable Accelerator for Keyword Spotting Based on Small-Footprint Temporal Efficient Neural Network”. en. In: *Electronics* 11.16 (Jan. 2022). Number: 16 Publisher: Multidisciplinary Digital Publishing Institute, p. 2571. ISSN: 2079-9292. DOI: 10.3390/electronics11162571. URL: <https://www.mdpi.com/2079-9292/11/16/2571> (visited on 07/13/2024).
- [31] Yu-Hsiang Huang, Pei-Hsuan Kuo, and Juinn-Dar Huang. “Hardware-Friendly Activation Function Designs and Its Efficient VLSI Implementations for Transformer-Based Applications”. In: *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. ISSN: 2834-9857. June 2023, pp. 1–5. DOI: 10.1109/AICAS57966.2023.10168591. URL: <https://ieeexplore.ieee.org/document/10168591> (visited on 07/13/2024).
- [32] Vikram Jain et al. “TinyVers: A Tiny Versatile System-on-Chip With State-Retentive eMRAM for ML Inference at the Extreme Edge”. In: *IEEE Journal of Solid-State Circuits* 58.8 (Aug. 2023). Conference Name: IEEE Journal of Solid-State Circuits, pp. 2360–2371. ISSN: 1558-173X. DOI: 10.1109/JSSC.2023.3236566. URL: <https://ieeexplore.ieee.org/abstract/document/10022047> (visited on 08/05/2024).
- [33] Jeff Johnson. *Rethinking floating point for deep learning*. arXiv:1811.01721 [cs]. Nov. 2018. DOI: 10.48550/arXiv.1811.01721. URL: <http://arxiv.org/abs/1811.01721> (visited on 03/04/2024).

- [34] Norman P. Jouppi et al. “In-Datcenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10 . 1145 / 3079856 . 3080246. URL: <https://doi.org/10.1145/3079856.3080246> (visited on 07/07/2024).
- [35] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. “An empirical exploration of recurrent network architectures”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, July 2015, pp. 2342–2350. (Visited on 07/01/2024).
- [36] Sehoon Kim et al. “I-BERT: Integer-only BERT Quantization”. In: *ArXiv* (Jan. 2021). URL: <https://www.semanticscholar.org/paper/I-BERT-Integer-only-BERT-Quantization-Kim-Gholami/7b8f3f65a98340d6e5ab94bd9a4ccb8f75704fd8> (visited on 07/06/2024).
- [37] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper*. arXiv:1806.08342 [cs, stat]. June 2018. DOI: 10.48550/arXiv.1806.08342. URL: <http://arxiv.org/abs/1806.08342> (visited on 07/22/2024).
- [38] Ulrich Kulisch. “Chapter 8 Scalar Products and Complete Arithmetic”. en. In: *Chapter 8 Scalar Products and Complete*. De Gruyter, Aug. 2008, pp. 245–301. ISBN: 978-3-11-020319-6. DOI: 10 . 1515 / 9783110203196 . 2 . 245. URL: <https://www.degruyter.com/document/doi/10.1515/9783110203196.2.245/html> (visited on 07/27/2024).
- [39] H. T. Kung and C. Leiserson. “Systolic Arrays for (VLSI).” In: Dec. 1978. URL: [https://www.semanticscholar.org/paper/Systolic-Arrays-for-\(VLSI\)-Kung-Leiserson/fbdf08fb10d92518b7c0571e1ae907de2c20ff76](https://www.semanticscholar.org/paper/Systolic-Arrays-for-(VLSI)-Kung-Leiserson/fbdf08fb10d92518b7c0571e1ae907de2c20ff76) (visited on 07/07/2024).
- [40] Andrey Kuzmin et al. “FP8 Quantization: The Power of the Exponent”. en. In: *Advances in Neural Information Processing Systems* 35 (Dec. 2022), pp. 14651–14662. URL: https://proceedings.neurips.cc/paper_files/paper/2022/hash/5e07476b6bd2497e1fbd11b8f0b2de3c-Abstract-Conference.html (visited on 06/08/2024).

- [41] Yanjing Li et al. "Q-ViT: Accurate and Fully Quantized Low-bit Vision Transformer". In: (2022). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2210.06707. URL: <https://arxiv.org/abs/2210.06707> (visited on 07/06/2024).
- [42] Yuhang Li et al. "BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction". In: *ArXiv* (Feb. 2021). URL: <https://www.semanticscholar.org/paper/BRECQ%3A-Pushing-the-Limit-of-Post-Training-by-Block-Li-Gong/6dffdeb81ebc761a8cce1e36b8d140d5b8d2a0e3> (visited on 07/06/2024).
- [43] Shih-Chii Liu et al. "Energy-efficient activity-driven computing architectures for edge intelligence". In: *2022 International Electron Devices Meeting (IEDM)*. ISSN: 2156-017X. Dec. 2022, pp. 21.2.1–21.2.4. DOI: 10.1109/IEDM45625.2022.10019443. URL: https://ieeexplore.ieee.org/abstract/document/10019443?casa_token=KFfZxqSLxLgAAAAA:FdEESFjTRreTBqB2-ZdrZ618kKPX0JVb8N1GsdYbChpQMFdq2Shh07H31SadoKWF0I68AVk (visited on 11/03/2023).
- [44] Siyuan Lu et al. "Hardware Accelerator for Multi-Head Attention and Position-Wise Feed-Forward in the Transformer". In: *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*. ISSN: 2164-1706. Sept. 2020, pp. 84–89. DOI: 10.1109/SOCC49529.2020.9524802. URL: <https://ieeexplore.ieee.org/abstract/document/9524802> (visited on 07/13/2024).
- [45] Loren Lugosch et al. *Speech Model Pre-training for End-to-End Spoken Language Understanding*. arXiv:1904.03670 [cs, eess]. July 2019. DOI: 10.48550/arXiv.1904.03670. URL: <http://arxiv.org/abs/1904.03670> (visited on 08/06/2024).
- [46] Alberto Marchisio et al. "SwiftTron: An Efficient Hardware Accelerator for Quantized Transformers". In: *2023 International Joint Conference on Neural Networks (IJCNN)* (June 2023). Conference Name: 2023 International Joint Conference on Neural Networks (IJCNN) ISBN: 9781665488679 Place: Gold Coast, Australia Publisher: IEEE, pp. 1–9. DOI: 10.1109/IJCNN54540.2023.10191521. URL: <https://ieeexplore.ieee.org/document/10191521/> (visited on 03/08/2024).

- [47] William Merrill, Jackson Petty, and Ashish Sabharwal. *The Illusion of State in State-Space Models*. arXiv:2404.08819 [cs]. June 2024. DOI: 10.48550/arXiv.2404.08819. URL: <http://arxiv.org/abs/2404.08819> (visited on 07/22/2024).
- [48] William Merrill et al. *A Formal Hierarchy of RNN Architectures*. arXiv:2004.08500 [cs]. Sept. 2020. DOI: 10.48550/arXiv.2004.08500. URL: <http://arxiv.org/abs/2004.08500> (visited on 10/12/2023).
- [49] Thorsten Müller and Dominik Kreutz. *Thorsten-Voice Dataset 2021.02*. deu. Feb. 2021. DOI: 10.5281/zenodo.5525342. URL: <https://zenodo.org/records/5525342> (visited on 08/06/2024).
- [50] Hubert Nguyen. “Parallel Prefix Sum (Scan) with CUDA”. In: *GPU Gems 3*. Addison-Wesley Professional, July 2007. ISBN: 978-0-321-51526-1.
- [51] Antonio Orvieto et al. *Resurrecting Recurrent Neural Networks for Long Sequences*. arXiv:2303.06349 [cs]. Mar. 2023. DOI: 10.48550/arXiv.2303.06349. URL: <http://arxiv.org/abs/2303.06349> (visited on 10/02/2023).
- [52] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. en. In: ().
- [53] Leonardo Ravaglia et al. “A TinyML Platform for On-Device Continual Learning With Quantized Latent Replays”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (Dec. 2021). Conference Name: IEEE Journal on Emerging and Selected Topics in Circuits and Systems, pp. 789–802. ISSN: 2156-3365. DOI: 10.1109/JETCAS.2021.3121554. URL: <https://ieeexplore.ieee.org/document/9580920/?arnumber=9580920> (visited on 07/08/2024).
- [54] Bitu Darvish Rouhani et al. *Microscaling Data Formats for Deep Learning*. arXiv:2310.10537 [cs]. Oct. 2023. DOI: 10.48550/arXiv.2310.10537. URL: <http://arxiv.org/abs/2310.10537> (visited on 07/27/2024).
- [55] Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. *Simplified State Space Layers for Sequence Modeling*. arXiv:2208.04933 [cs]. Mar. 2023. DOI: 10.48550/arXiv.2208.04933. URL: <http://arxiv.org/abs/2208.04933> (visited on 02/17/2024).

- [56] Vivienne Sze et al. “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”. In: *Proceedings of the IEEE* 105.12 (Dec. 2017). Conference Name: Proceedings of the IEEE, pp. 2295–2329. ISSN: 1558-2256. DOI: 10 . 1109 / JPROC . 2017 . 2761740. URL: <https://ieeexplore.ieee.org/abstract/document/8114708> (visited on 07/30/2024).
- [57] Yi Tay et al. *Long Range Arena: A Benchmark for Efficient Transformers*. arXiv:2011.04006 [cs]. Nov. 2020. DOI: 10 . 48550/arXiv . 2011 . 04006. URL: <http://arxiv.org/abs/2011.04006> (visited on 07/22/2024).
- [58] Neil Thompson et al. “The Computational Limits of Deep Learning”. en. In: *Computing within Limits*. LIMITS, June 2023. DOI: 10 . 21428/bf6fb269 . 1f033948. URL: <https://limits.pubpub.org/pub/wm1lwjce/release/1> (visited on 07/01/2024).
- [59] Tsung-Han Tsai and Ding-Bang Lin. “An On-Chip Fully Connected Neural Network Training Hardware Accelerator Based on Brain Float Point and Sparsity Awareness”. In: *IEEE Open Journal of Circuits and Systems* 4 (2023). Conference Name: IEEE Open Journal of Circuits and Systems, pp. 85–98. ISSN: 2644-1225. DOI: 10 . 1109 / OJCAS . 2023 . 3245061. URL: <https://ieeexplore.ieee.org/abstract/document/10051716> (visited on 11/04/2023).
- [60] Ivan Tsmots, Oleksa Skorokhoda, and Vasyl Rabyk. “Hardware Implementation of Sigmoid Activation Functions using FPGA”. In: *2019 IEEE 15th International Conference on the Experience of Designing and Application of CAD Systems (CADSM)*. ISSN: 2572-7591. Feb. 2019, pp. 34–38. DOI: 10 . 1109 / CADSM . 2019 . 8779253. URL: <https://ieeexplore.ieee.org/document/8779253> (visited on 07/29/2024).
- [61] Yohann Uguen and Florent de Dinechin. “Design-space exploration for the Kulisch accumulator”. Mar. 2017. URL: <https://hal.science/hal-01488916> (visited on 08/05/2024).
- [62] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html> (visited on 07/01/2024).
- [63] Naigang Wang et al. *Training Deep Neural Networks with 8-bit Floating Point Numbers*. arXiv:1812.08011 [cs, stat]. Dec. 2018. DOI: 10 . 48550/arXiv . 1812 . 08011. URL: <http://arxiv.org/abs/1812.08011> (visited on 02/07/2024).

- [64] Pete Warden. *Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition*. arXiv:1804.03209 [cs]. Apr. 2018. DOI: 10.48550/arXiv.1804.03209. URL: <http://arxiv.org/abs/1804.03209> (visited on 08/06/2024).
- [65] Xiuying Wei et al. “QDrop: Randomly Dropping Quantization for Extremely Low-bit Post-Training Quantization”. In: (2022). Publisher: arXiv Version Number: 2. DOI: 10.48550/ARXIV.2203.05740. URL: <https://arxiv.org/abs/2203.05740> (visited on 07/06/2024).
- [66] P.J. Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (Oct. 1990). Conference Name: Proceedings of the IEEE, pp. 1550–1560. ISSN: 1558-2256. DOI: 10.1109/5.58337. URL: https://ieeexplore.ieee.org/abstract/document/58337?casa_token=fBzmXfKS-W4AAAAA:WNpo09CfyiPTzM860tPehBmN1fR2ILR42frGkqKxpysH7AQSg8-N9513w494JgUUAk6fKco (visited on 10/23/2023).
- [67] Jin Xu et al. “Reluplex made more practical: Leaky ReLU”. In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. ISSN: 2642-7389. July 2020, pp. 1–7. DOI: 10.1109/ISCC50000.2020.9219587. URL: <https://ieeexplore.ieee.org/abstract/document/9219587> (visited on 07/29/2024).
- [68] Junichi Yamagishi, Christophe Veaux, and Kirsten MacDonald. “CSTR VCTK Corpus: English Multi-speaker Corpus for CSTR Voice Cloning Toolkit (version 0.92)”. eng. In: *The Rainbow Passage which the speakers read out can be found in the International Dialects of English Archive: (<http://web.ku.edu/~ideal/readings/rainbow.htm>)*. (Nov. 2019). Accepted: 2019-11-13T17:09:33Z Publisher: University of Edinburgh. The Centre for Speech Technology Research (CSTR). DOI: 10.7488/ds/2645. URL: <https://datashare.ed.ac.uk/handle/10283/3443> (visited on 08/06/2024).
- [69] Jinsung Yoon et al. “A Real-Time Keyword Spotting System Based on an End-To-End Binary Convolutional Neural Network in FPGA”. In: *2023 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. ISSN: 2473-4683. Apr. 2023, pp. 1–3. DOI: 10.1109/COOLCHIPS57690.2023.10121981. URL: <https://ieeexplore.ieee.org/abstract/document/10121981> (visited on 08/05/2024).
- [70] Juyoung Yun, Byungkong Kang, and Zhoulai Fu. *The Hidden Power of Pure 16-bit Floating-Point Neural Networks*. arXiv:2301.12809 [cs]. Jan.

2023. DOI: 10.48550/arXiv.2301.12809. URL: <http://arxiv.org/abs/2301.12809> (visited on 02/05/2024).
- [71] Pedram Zamirai et al. *Revisiting BFloat16 Training*. arXiv:2010.06192 [cs, stat]. Mar. 2021. DOI: 10.48550/arXiv.2010.06192. URL: <http://arxiv.org/abs/2010.06192> (visited on 02/08/2024).
- [72] Andreas Zell. *Simulation neuronaler netze*. Vol. 1. 5.3. Addison-Wesley Bonn, 1994. URL: <https://doc1.bibliothek.li/aal/FLMF007250.pdf> (visited on 11/15/2023).
- [73] Yichi Zhang et al. *Binarized Neural Machine Translation*. arXiv:2302.04907 [cs]. Feb. 2023. DOI: 10.48550/arXiv.2302.04907. URL: <http://arxiv.org/abs/2302.04907> (visited on 03/12/2024).
- [74] Shuchang Zhou et al. “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients”. In: *ArXiv* (June 2016). URL: [https://www.semanticscholar.org/paper/DoReFa-Net % 3A - Training - Low - Bitwidth - Convolutional - Low - Zhou - Ni / 8b053389eb8c18c61b84d7e59a95cb7e13f205b7](https://www.semanticscholar.org/paper/DoReFa-Net-%20-%20Training-Low-Bitwidth-Convolutional-Low-Zhou-Ni/8b053389eb8c18c61b84d7e59a95cb7e13f205b7) (visited on 07/06/2024).
- [75] Feng Zhu et al. “Towards Unified INT8 Training for Convolutional Neural Network”. In: 2020, pp. 1969–1979. URL: https://openaccess.thecvf.com/content_CVPR_2020/html/Zhu_Towards_Unified_INT8_Training_for_Convolutional_Neural_Network_CVPR_2020_paper.html (visited on 12/07/2023).
- [76] Nicolas Zucchet et al. *Online learning of long-range dependencies*. arXiv:2305.15947 [cs]. May 2023. DOI: 10.48550/arXiv.2305.15947. URL: <http://arxiv.org/abs/2305.15947> (visited on 09/22/2023).