IEEE Open Journal of
Circuits and Systems

Received 19 November 2024; revised 8 January 2025 and 30 January 2025; accepted 23 February 2025. Date of publication 26 February 2025; date of current version 17 March 2025.

Digital Object Identifier 10.1109/OJCAS.2025.3546067

NLU: An Adaptive, Small-Footprint, Low-Power Neural Learning Unit for Edge and IoT Applications

AMIRHOSSEIN ROSTAMI[1,2], SEYED MOHAMMAD ALI ZEINOLABEDIN[3], LIYUAN GUO[1,2], FLORIAN KELBER[1], HEINER BAUER[1], ANDREAS DIXIUS[1], STEFAN SCHOLZE[1], MARC BERTHEL[1], DENNIS WALTER[1,4], JOHANNES UHLIG[1], BERNHARD VOGGINGER[1], AND CHRISTIAN MAYR[1,2,4] (Senior Member, IEEE)

[1]Faculty of Electrical and Computer Engineering, Technische Universität Dresden, 01069 Dresden, Germany
[2]Center for Scalable Data Analytics and Artificial Intelligence, 01062 Dresden/Leipzig, Germany
[3]Blackrock Neurotech, Salt Lake City, UT 84108, USA
[4]Centre for Tactile Internet With Human-in-the-Loop, TU Dresden, 01069 Dresden, Germany
This article was recommended by Associate Editor Y. Du.

CORRESPONDING AUTHOR: A. ROSTAMI (e-mail: Amirhossein.Rostami@tu-dresden.de)

This work was supported in part by the ECSEL Joint Undertaking (JU) under Grant 876925, which receives support from the European Union's Horizon 2020 Research and Innovation Programme and France, Belgium, Germany, The Netherlands, Portugal, Spain, and Switzerland; in part by the German Research Foundation, Deutsche Forschungsgemeinschaft (DFG) as part of Germany's Excellence Strategy–EXC 2050/1– Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden under Project 390696704; in part by the German Federal Ministry of Education and Research (BMBF) and the Free State of Saxony within the ScaDS.AI Center of Excellence for AI Research; in part by the German Federal Ministry of Education and Research (BMBF) within the project "Pattern-Recognition for In-vitro Signal analysis Through fully Integrated NeuroElectronics - ZEN2_PRISTINE" under Grant 16ME0380K.

ABSTRACT Over the last few years, online training of deep neural networks (DNNs) on edge and mobile devices has attracted increasing interest in practical use cases due to their adaptability to new environments, personalization, and privacy preservation. Despite these advantages, online learning on resource-restricted devices is challenging. This work demonstrates a 16-bit floating-point, flexible, power- and memory-efficient neural learning unit (NLU) that can be integrated into processors to accelerate the learning process. To achieve this, we implemented three key strategies: a dynamic control unit, a tile allocation engine, and a neural compute pipeline, which together enhance data reuse and improve the flexibility of the NLU. The NLU was integrated into a system-on-chip (SoC) featuring a 32-bit RISC-V core and memory subsystems, fabricated using GlobalFoundries 22nm FDSOI technology. The design occupies just $0.015mm^2$ of silicon area and consumes only 0.379 mW of power. The results show that the NLU can accelerate the training process by up to 24.38× and reduce energy consumption by up to 37.37× compared to a RISC-V implementation with a floating-point unit (FPU). Additionally, compared to the state-of-the-art RISC-V with vector coprocessor, the NLU achieves 4.2× higher energy efficiency (measured in GFLOPS/W). These results demonstrate the feasibility of our design for edge and IoT devices, positioning it favorably among state-of-the-art on-chip learning solutions. Furthermore, we performed mixed-precision on-chip training from scratch for keyword spotting tasks using the Google Speech Commands (GSC) dataset. Training on just 40% of the dataset, the NLU achieved a training accuracy of 89.34% with stochastic rounding.

INDEX TERMS Online learning, on-device learning, on-chip training, deep neural network (DNN), application specific integrated circuit (ASIC), co-design, energy efficient, hardware accelerator, reduced precision computation, bfloat16.

I. INTRODUCTION

THE FIELD of artificial intelligence (AI) has gained significant attention in recent years, driven by groundbreaking advancements in deep neural networks (DNNs), the abundance of data, and the increasing computational power. AI has extended its reach into diverse

domains, including areas such as drug discovery, medical diagnosis, recommendation systems, and natural language processing [1], [2], [3], [4], [5]. Due to the intensive computing demands of DNNs, the need to accelerate algorithms has become crucial for both cloud servers and edge devices [6].

DNNs involve two main stages: training, where the network parameters (weights) are tuned based on data to learn a task, and inference, where the trained network makes predictions based on new data [7]. Deep learning models are typically trained in the cloud using high-end processors such as NVIDIA A100 GPUs, Google TPUs, and AWS Trainium. The pre-trained models are then deployed on the cloud using Xilinx Virtex UltraScale+ FPGAs, Intel Xeon Scalable CPUs, and AWS Inferentia, or are applied to edge and tiny devices for inference applications [6], [8], [9], [10], [11], [12]. Edge computing offers several advantages over cloud computing for certain applications. It is particularly beneficial when working with private data, such as medical records, when AI needs to interact in real-time, as in robotic surgery, or in remote locations where communication infrastructures are unavailable. Additionally, in wearable devices, eliminating communication with cloud servers can reduce power consumption and extend battery life [13], [14], [15], [16], [17], [18].

Despite these benefits, edge devices have limited memory and computational power. To address this, many edge-friendly deep learning models have been proposed for Tiny Machine Learning (TinyML), such as XiNet [19], EtinyNet [20], and MCUNet [12]. Additionally, hardware solutions like TinyVers [21], SushiAccel [22], and AccelTran [23], have been developed for inference applications. However, the accuracy of these systems may decline over time, as they cannot adapt to new environments and user data. This problem is usually known as inference degradation [24], [25]. Therefore, it is important to update the DNN models with new data to maintain their performance.

To mitigate inference degradation, updating DNN models with new data is crucial. Due to either a lack of communication infrastructure or privacy concerns, it is not always possible to retrain the models on the cloud and deploy them on edge devices [17]. To address privacy concerns, federated learning, in which several devices share only network parameters and not private user data was proposed in [26]. However, training DNN models on edge devices with limited memory and computational power for online and continual learning remains a challenging and active area of research [27]. Further advances in algorithm development are still necessary to fully unlock the potential of learning on the edge.

On-device training offers several advantages: it enables continuous adaptation to new data, preserves privacy, ensures real-time operation, and eliminates reliance on communication infrastructure. Existing hardware accelerators for DNNs can be broadly classified into two categories: General matrix multiplication (GEMM) accelerators and Network-specific accelerators. Previous studies have profiled various DNN models, typically using large batch sizes, and found that the majority of computation time is spent on General Matrix Multiplications (GEMMs) [28], [29], [30]. Consequently, many hardware systems have been developed to accelerate GEMM operations [27], [30], [31], [32], [33]. Typically, a systolic array is used to accelerate GEMMs [30], [31], [32], [34]. Compared to other approaches such as general-purpose matrix processing arrays (e.g., those relying on centralized control or broadcasting techniques), systolic arrays can achieve better data reuse, allowing for more operations to be applied to the data before it is written back to memory [32]. However, this approach has two main issues. Firstly, compared to general-purpose matrix processing arrays, systolic arrays contain more computation units. Therefore, they occupy more silicon area and consume more energy, making them unsuitable for some applications [12], [35]. Secondly, while systolic arrays perform well in matrix multiplications, DNN operations extend beyond just matrix multiplications. They also involve elementwise scalar, vector, and matrix operations, as well as non-linear activation functions. For these kinds of operations, systolic arrays heavily depend on a processor, and they cannot work independently. Matrix multiplication dominates DNN operations when the batch size is large. However, in online learning with a batch size of one, other operations become more important. Furthermore, in the context of edge devices and multi-core systems where limited memory and computation are available for each core, deploying a large batch size (usually more than one) is not practical [12].

On the other hand, network-specific accelerators are designed for particular neural networks and layers. These designs, with fixed architectures, can deliver high efficiency and performance for their targeted neural network types. However, this approach is limited to specific layers, lacks generalizability to other or newer networks, is not scalable, and may quickly become obsolete as neural network architectures evolve. Furthermore, the optimization techniques applied in these designs are typically valid for only one network and one dataset, further restricting their utility [36], [37], [38], [39], [40], [41].

Motivated by these challenges, this paper introduces a novel, efficient, and highly configurable general-purpose 16-bit floating-point hardware accelerator, referred to as the Neural Learning Unit (NLU). The NLU is designed to integrate online learning capabilities into devices with limited memory, such as edge devices or many-core systems. Unlike GEMM accelerators, the NLU supports a wide range of scalar, vector, and matrix operations. Furthermore, unlike specialized architectures, it is adaptable to various neural networks, including recurrent neural networks (RNNs), spiking RNNs (SRNNs), and multilayer perceptrons (MLPs), the latter serving as the foundational model for well-known transformer networks. The NLU can be configured either during the startup process or dynamically reconfigured at runtime. Once configured, it is capable of functioning independently and performing complex computations.

Our main contributions to this work are summarized as follows:

1) We propose a novel reconfigurable Neural Learning Unit (NLU) that integrates a dynamic control unit, tile allocation engine, and neural compute pipeline. This design supports a runtime-configurable architecture, minimizes data movement, and significantly enhances performance for on-chip DNN training.
2) We achieve up to 24.38× faster training and up to 37.37× energy savings compared to a standard RISC-V with a floating-point hardware accelerator solution. These improvements are validated on workloads representative of online and edge-level learning scenarios.
3) We demonstrate real-world feasibility by training a three-layer network for keyword spotting tasks entirely on-chip using only 40% of the training dataset, reaching 89.34% accuracy with a bfloat16 precision format.
4) We validate the hardware implementation in silicon, fabricated in GlobalFoundries 22nm FDSOI technology, confirming the low power consumption and small footprint of our design.

The remainder of this article is structured as follows: Section II presents the DNN training process, explores previous hardware for training DNN models, and profiles the training workloads for finding primitive operations. Section III introduces the operations and explains how to accelerate them using eight FMA (fused-multiply-add) units. Section IV proposes NLU, a novel hardware for training DNN models. NLU connects to an instruction memory and can compute complex combinations of scalar, vector, and matrix operations as well as ReLU and STEP nonlinear activation functions. This section explains its functionality and the block and tile mechanism in detail. Section V evaluates the hardware and compares it with the state-of-the-art hardware. Finally, Section VI wraps up this work and suggests ideas for future studies.

II. BACKGROUND AND MOTIVATIONS

In the previous section, we introduced the growing need for efficient on-device training solutions and outlined the key challenges arising from limited memory and compute resources on edge devices. In this section, we build upon those challenges by reviewing relevant work on hardware designs for training deep neural networks and discussing why an adaptable accelerator design is necessary. Furthermore, we explore existing data formats, network architectures, and profiling analyses to clarify the motivations behind our proposed Neural Learning Unit (NLU).

A. DNN TRAINING

The training process involves three main stages: the Forward pass, Backward pass, and optimization. In the Forward pass, the model applies input data to randomly initialized network parameters (weights) and computes the error between the prediction and the ground truth label. During the Backward pass, the gradients of weights with respect to the loss function are computed. In the optimization step, the weights are updated based on the gradients to reduce the error. This process is repeated for all inputs in the dataset over multiple epochs, continuing until the accuracy reaches an acceptable level [17].

B. RELATED WORKS

1) HARDWARE

Recently, many AI hardware, accelerators, and processor extensions have been introduced to speed up the training process [30], [31], [32], [33], [38], [42], [43], [44], [45], [46], [47]. We can divide them into two main categories: domain-specific accelerators (DSA) and neural processing units (NPU).

DSAs are specialized accelerators designed to speed up a specific operation, typically matrix multiplication. As examples, systolic arrays are used by Google TPU [43], [44], SIGMA [30], Gamma [42], SARA [32], VEGETA [31] and RedMule [33] to accelerate GEMM operations.

NPUs are systems on chips (SoCs) that comprise processing elements (PEs) and Networks on Chips (NoCs). Additionally, they feature PCI Express or Ethernet connections to the host for configurations or data movements. The PE includes a Vector Processing Unit (VPU) for executing vector and matrix operations, a Specific Function Unit (SFU) for non-linear activation functions, and internal memory for storing network parameters, intermediate results, and datasets. NoCs provide adaptive links to connect PEs and establish a dataflow graph in hardware, similar to software. IBM [45], SambaNova [46], and Tenstorrent [47] chips belong to this category.

This paper focuses on designing a PE that can be utilized in NPU systems because, compared to DSAs, NPUs are general-purpose and support more applications. The PE can serve as an online learning accelerator for a processor in edge applications, a device in federated learning, or a training accelerator for many-core systems. In this scenario, a large batch of data is distributed among processing elements (PEs), with one or several PEs handling computations for one batch.

2) DATA FORMAT

Over the past decades, the model size has increased exponentially [48]. From a hardware standpoint, this expansion demands more memory and computational resources. Training is typically done in a 32-bit floating-point format. Previous studies have shown that lower precision data formats can also be employed in the training process. In [49], float16 (half precision floating point), in [50], bfloat16 (Brain Floating Point), and in [51], DLFloat (Deep Learning Floating Point) data formats have been explored. For example, in [50], findings reveal that training AlexNet with bfloat16 resulted in a 0.6% drop in top-5 accuracy compared to float32, while for ResNet, it achieved the same accuracy.

For this study, we chose the bfloat16 data format because, unlike float16, it supports a larger data range and does not require additional hyperparameter tuning (loss factor). The conversion between 32-bit and 16-bit is straightforward and does not demand sophisticated hardware. Additionally, bfloat16 is evaluated in various commercial chips, like Google TPU.

C. PROFILING

In this section, we aim to identify primitive operations in online learning (when the batch size is one) and design a flexible accelerator to enhance their execution. In this scenario, the accelerator can support various algorithms and is not limited to a specific DNN model.

To reach this goal, we analyzed MLP [52], GRU [53], and SRNN [54], [55] algorithms using a batch size of one. The configurations for each network are as follows:

- *MLP:* The multilayer perceptron consists of 72 input neurons, 72 hidden neurons, and 24 output neurons. The first layer uses the ReLU activation function, while the second layer employs SoftMax. Backpropagation was utilized to train the network.
- *GRU:* The gated recurrent unit network comprises 80 input neurons, 24 recurrent neurons, and 24 output neurons. Backpropagation through time (BPTT) was used for training. To optimize the backward pass, activation values calculated during the forward pass were reused for gradient computation, eliminating redundant calculations.
- *SRNN:* The spiking recurrent neural network consists of 80 input neurons, 24 spiking recurrent neurons, and 16 output neurons. This network was trained using e-prop [54], [55], a biologically inspired learning rule that leverages spiking neuron dynamics.

C code for these algorithms was developed and executed on an AMD Ryzen™ 7 PRO CPU without utilizing any hardware acceleration. For this purpose, the forward and backward passes were considered. The operations in the optimization process depend on the algorithm (gradient descent, RMSProp, ADAM, etc.), and for recurrent networks, the optimization process is executed after the last time step, which, compared to other tasks, occupies a small portion of processor time. Additionally, the optimization process is usually conducted in the float32 data format because it needs more precision [49].

We divided the operations into four groups: Vector-Matrix-Multiply, Elementwise operations, Outer product, and special functions. Elementwise operations consist of scalar, vector, and matrix operations such as multiplication, addition, and subtraction. The outer product involves two vectors to produce a matrix. The specific functions group comprises nonlinear activation functions such as ReLU, tanh, sigmoid, and softmax that are usually applied to a data vector.

Figure 1 displays the profiling results. As shown in the figure, for MLP, Vector-Matrix-Multiply and outer product occupy most of the CPU time. For GRU and SRNN, CPU time is distributed among Elementwise, Vector-Matrix-Multiply, and Outer product operations. We aim to design a hardware accelerator to speed up these operations and ReLu and STEP activation functions.

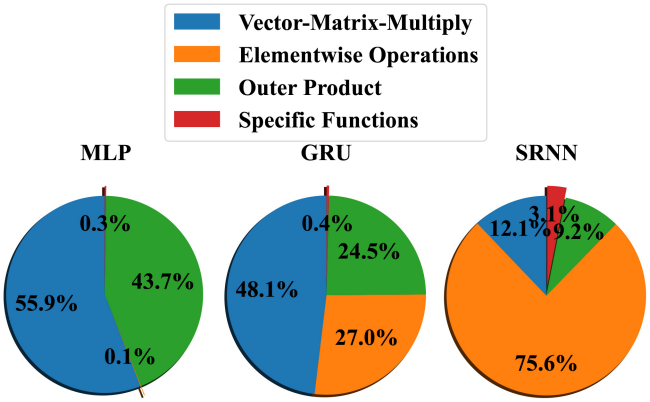


FIGURE 1. Profiling results based on the CPU runtime of primitive operations for training MLP, GRU, and SRNN networks when the batch size is one (online gradient descent).

D. CHALLENGES

Profiling results show that the accelerator should support many diverse operations. There are two main challenges in designing a hardware accelerator to support diverse operations, such as vector-matrix multiplication and elementwise scalar, vector, and matrix operations.

The first challenge is that traditional approaches are not memory-efficient. For intermediate results, especially in the case of elementwise operations, one vector or matrix must be computed completely, requiring additional memory space to store the intermediate results. This memory space may not be available, particularly when computing large matrices [14], [17]. Quantitatively, for a matrix size of 128×128, intermediate results alone can require 16 KB of memory, which may exceed the available buffer capacity of low-power accelerators designed for edge applications.

The second challenge is that traditional approaches reduce acceleration. Typically, the processor configures the accelerator for each operation and waits until the operation finishes before configuring it for the next one [32], [39]. In this case, the processor is occupied with configuring the accelerator, and the accelerator must wait for the new configuration, resulting in reduced speed-up. For instance, in systems where the processor takes 10 clock cycles for configuration and the accelerator requires 20 cycles per operation, the overall throughput may drop by over 30% due to communication delays.

In summary, Section II examined the fundamental challenges of on-device DNN training. The following section details how we address these challenges, first by examining the primitive operations and then by presenting our proposed accelerator architecture.

III. OPERATIONS

Building on the workload analysis in Section II, we focus here on the core operations that dominate on-device DNN training with batch size one. As revealed in our profiling (Figure 1), elementwise arithmetic, vector–matrix multiplication, and outer products account for most computational demands when training models such as MLPs, GRUs, and SRNNs. In this section, we detail how these operations can be systematically accelerated, forming the basis for our proposed hardware design in Section IV.

The accelerator can connect to a data bus of arbitrary size. For this study, we plan to connect the accelerator to a 128-bit data bus. In every memory access, eight 16-bit data units are transferred. We plan to use eight FMA units. A lower number of FMA units increases latency, and a higher number of FMA units requires more silicon area and consumes more power, which is suboptimal.

In this section, we examine the operations in more detail and show how, by using eight FMA units, we can execute these operations.

A. ELEMENT-WISE AND OUTER PRODUCT OPERATIONS

Element-wise Operations can be a mixture of scalar, vector, and matrix additions, subtractions, and multiplications. To make it clearer, consider the following example that contains a variety of operations (Figure 2):

$$m1_{ij} = m1_{ij} + (K \times v1_i) \times \left(v2_i \times \left(h1_j - h2_j\right)\right) \tag{1}$$

where i belongs to $\{1, \ldots, N_{row}\}$, j belongs to $\{1, \ldots, N_{col}\}$, m1 is a $[N_{row}] \times [N_{col}]$ matrix, K is a scalar constant, v1 and v2 is a $[N_{row}] \times [1]$ vector (column vector), h1 and h2 is a $[1] \times [N_{col}]$ vector (row vector). This equation is analogous to the eligibility vector computation in spiking recurrent neural networks (SRNN) [54], [55]. Moreover, similar element-wise operations are commonly found in other neural network layers such as the Gated Recurrent Unit (GRU), linear layers, and layer normalization in transformer architectures [56]. These operations are also present in optimization techniques like Adam and loss function computations such as cross-entropy. In this example N_{row} is 24 and N_{col} is 16. We are interested in computing all elements of $m1_{ij}$. One can divide this equation into simple operations that can be accelerated by hardware. As illustrated in Figure 2, these operations consist of column vector-vector subtraction, scalar-matrix multiplication, outer product, element-wise column vector-matrix multiplication, and finally, matrix-matrix addition. In this case, it requires extra arrays to store the intermediate results, as shown in the following equations:

$$\begin{aligned}
&(a)\ h3_j = h1_j - h2_j \\
&(b)\ v3_i = K \times v1_i \\
&(c)\ m2_{ij} = v2_i \otimes h3_j \\
&(d)\ m3_{ij} = v3_i \odot m2_{ij} \\
&(e)\ m1_{ij} = m1_{ij} + m3_{ij}
\end{aligned} \tag{2}$$

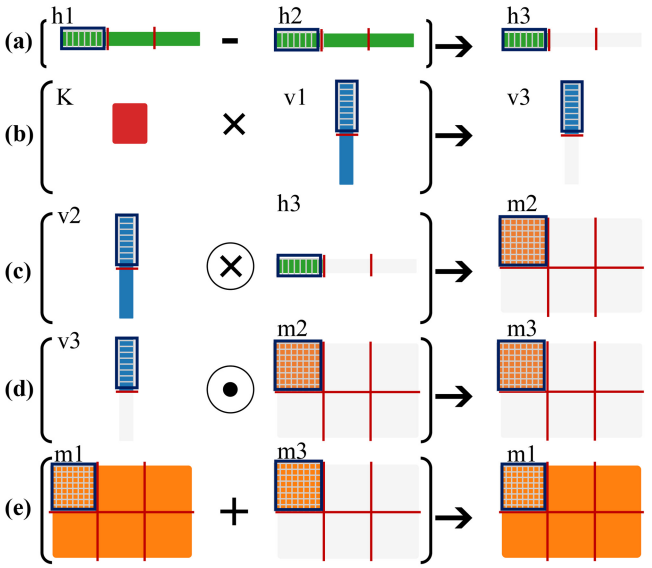


FIGURE 2. Example for element-wise operations: The ⊙ symbolizes element-wise multiplication, while the ⊗ denotes the outer product. (a) Two row vectors are subtracted. (b) A scalar is multiplied by a column vector. (c) A matrix is produced from the outer product. (d) A column vector is multiplied by every column of a matrix. (e) Two matrices are added. There is no need to store light gray arrays in the memory.

where the h3, v3, m2 and m3 are used to store the intermediate results.

To save memory and speed up the operations simultaneously, we tile the data. For vectors, one tile consists of eight 16-bit data elements and is available in a single memory read. For matrices, one tile consists of sixty-four 16-bit data elements and is available in eight memory reads. Storing only one tile for intermediate variables ($h3$, $v3$, $m2$, and $m3$) is sufficient, and there is no need to save all values. Figure 2 shows intermediate variables in light gray.

The computation is partitioned into several iterations depending on the matrix size. For example, in Figure 2 to compute $m1$, there are six iterations. In each iteration, the operations are executed on a single tile of operands, and only one tile of matrix $m1$ is computed. After completing six iterations, the final result becomes available.

Reusing the same memory for one tile can reduce extra memory usage further. In this example, using the traditional approach, we need 2,544 bytes of memory, but with our approach, we only need 1,216 bytes of memory for the computation. This means a reduction in memory usage of 52.2%.

B. VECTOR-MATRIX MULTIPLICATION

Vector-matrix Multiplications can be represented in four forms:

$$C^T_{[1]\times[N_{row}]} = A_{[1]\times[N_{col}]} \times B^T_{[N_{col}]\times[N_{row}]} \tag{3}$$

$$C_{[N_{row}]\times[1]} = B_{[N_{row}]\times[N_{col}]} \times A^T_{[N_{col}]\times[1]} \tag{4}$$

$$C_{[1]\times[N_{col}]} = A^T_{[1]\times[N_{row}]} \times B_{[N_{row}]\times[N_{col}]} \tag{5}$$

$$C^T_{[1]\times[N_{col}]} = B^T_{[N_{col}]\times[N_{row}]} \times A_{[N_{row}]\times[1]} \tag{6}$$

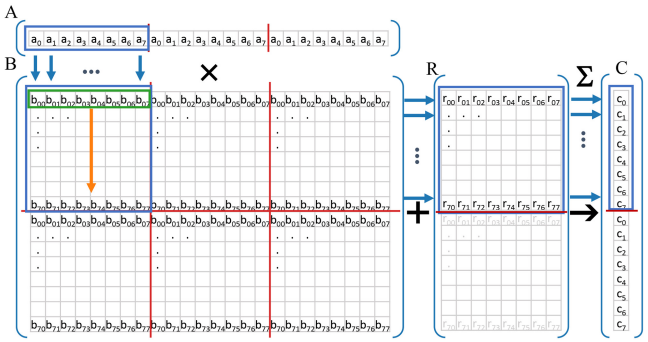


FIGURE 3. Vector Matrix Multiplication. It demonstrates how to accelerate vector-matrix multiplication by using FMA units. First, one tile of vector A and one tile of matrix B are read. Then, the vector is multiplied by each row of the matrix. The result is added to the output buffer. This operation is repeated for all tiles in one row tile. Afterward, the data in the output buffer are accumulated row-wise to calculate one tile of output vector C.

The results of (3) and (4) are equivalent. Also the results of (5) and (6) are similar. The (5) and (6) are actually the transpose version of (3) and (4). Usually, one form is needed in the forward pass and the transposed version of it is used in the backward pass.

If we have $C = \{c_0, \ldots, c_{N_{row}}\}$, $A = \{a_0, \ldots, a_{N_{col}}\}$ and $B = \{b_{00}, b_{01}, \ldots, b_{10}, b_{11}, \ldots, b_{N_{row}N_{col}}\}$,

(3) can be computed as:

$$c_i = \sum_{i=0}^{N_{row}} \sum_{j=0}^{N_{col}} \left(a_j \times b_{ij} + c_i\right) \tag{7}$$

To perform this computation efficiently using eight FMA units, we break down the equation into smaller tiles. Each tile comprises an 8×8 matrix and 8 vector elements. The rewritten form is as follows:

$$c_i = \sum_{i=1}^{\frac{N_{row}}{8}} \sum_{j=1}^{\frac{N_{col}}{8}} \sum_{k=1}^{8} \sum_{l=1}^{8} \left(a_{[8j+l]} \times b_{[8i+k][8j+l]} + c_{[8i+k]}\right) \tag{8}$$

First, we compute two inner summations ($R = A \times B + C$) for one tile, followed by two outer summations. In this case, the direction of computation matches the direction of tile movement, there is no need to store the intermediate result R in memory and read it again. After going through one row tile, we accumulate the R row-wise to produce the final result C and store the values in memory. Figure 3 illustrates this computation visually. Also, here, having eight FMA units for computations along with an 8×8 buffer array for storing intermediate results is sufficient for executing transpose vector-matrix multiplication.

Equation (5) can be computed similarly, but we need to accumulate the results along the column to obtain the final outcome. Since these data are not available until the computation of the last row tile, we need to store the intermediate results for each row tile and use them later.

C. SPECIAL FUNCTIONS

Nonlinear activation functions include various functions such as ReLU, STEP, exponent, tanh, sigmoid, and softmax. We implement two activation functions: ReLU and STEP (the gradient of ReLU). The equation for ReLU is as follows:

$$ReLU(x) = max(0, x) \tag{9}$$

For ReLU computation, one tile of input (8×8) is read from memory. The sign bit is checked, and if the number is negative, it will be replaced with zero and then written back to memory.

The equation for STEP is as follows:

$$STEP(x) = \begin{cases} 0 \ if \ x \leq 0 \\ 1 \ if \ x > 0 \end{cases} \tag{10}$$

For STEP function computation, one tile of input is read from memory. For each number, if the sign bit is zero and the exponent is non-zero, it means that the number is larger than zero, and it is replaced with one; otherwise, it is replaced with zero.

By combining nonlinear activation functions with elementwise scalar, vector, or matrix operations, it becomes possible to construct or approximate more complex functions. For example, in Section V part V-D, for on-device training, we utilize the function $f(x) = (1 - \frac{x^2}{2}) \cdot \text{STEP}(-x + 1.4) \cdot \text{STEP}(x + 1.4)$ to approximate a Gaussian function, $f(x) = e(-x^2)$.

In this study, we focused on implementing simpler nonlinear functions such as ReLU and STEP due to their lower hardware complexity and frequent usage in modern neural networks. Functions such as Softmax, Sigmoid, and Tanh require computationally intensive operations like division, exponentiation, and logarithms, which were excluded from the current design.

D. MATRIX-MATRIX MULTIPLICATION AND CONVOLUTION

To perform matrix-matrix multiplication, we can divide the first matrix into multiple vectors and use vector-matrix multiplication operation to compute the matrix-matrix multiplication. In addition by unrolling convolution and utilizing matrix-matrix multiplication, we can execute convolution [57], [58].

In summary, in this section, we examined the fundamental operations essential for on-device deep learning, including elementwise, outer product, and vector-matrix multiplications, as well as activation functions. These operations form the core building blocks of training workloads, particularly when the batch size is one. Next, we present how these operations are integrated into a specialized architecture. In the following section (Section IV), we introduce our proposed hardware design, showing how we implement these operations efficiently in silicon while minimizing data movement and enabling run-time reconfigurability.

IV. HARDWARE DESIGN

A. DESIGN DECISIONS

Building on the analysis of DNN workloads, we now present our hardware accelerator design, explaining how it overcomes the limitations identified earlier. The accelerator is

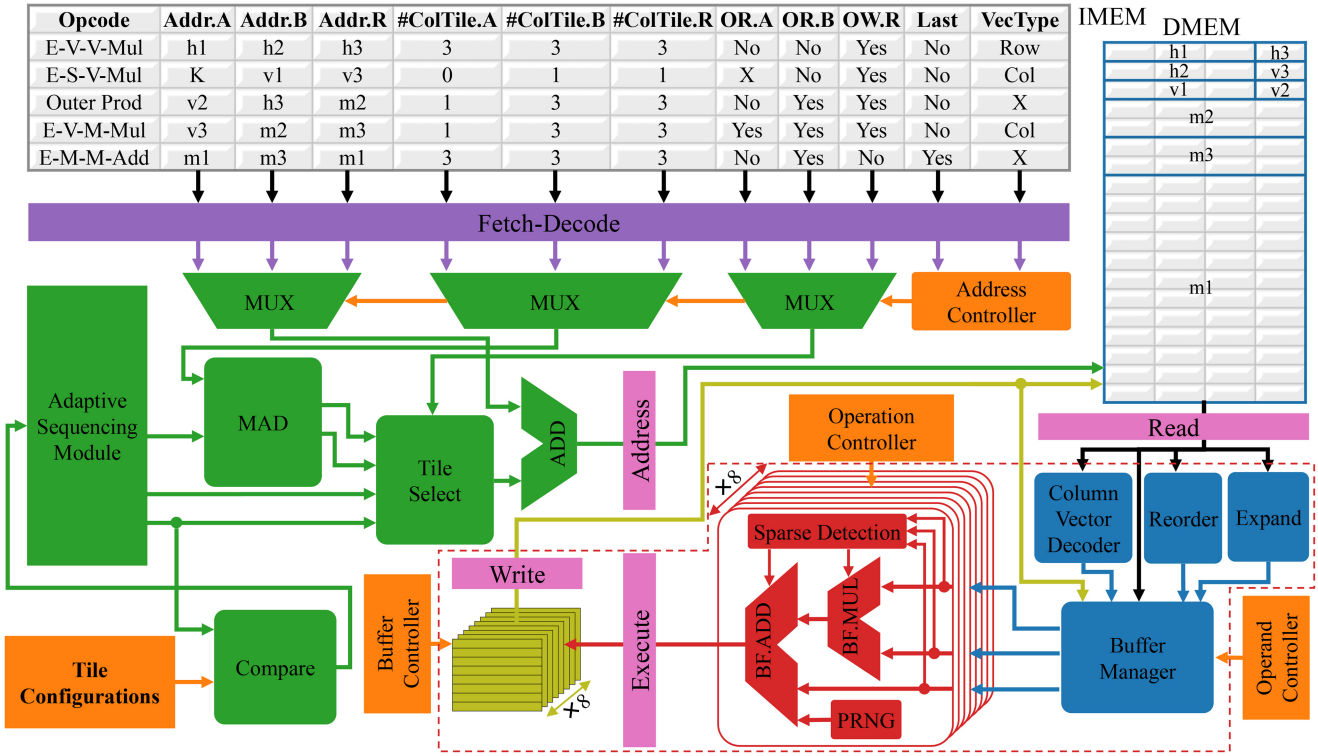


FIGURE 4. Details of the NLU architecture design: The top left shows the instruction, and the top right shows the data memory. The Tile Allocation Engine (TAE), is highlighted in green. The Neural Compute Pipeline (NCP), comprises the input pipeline buffer (in blue), the arithmetic logic unit (in red), and the output pipeline buffer (in yellow). The Dynamic Control Unit (DCU), responsible for runtime control, is depicted in orange.

designed to connect to different data bus widths to accommodate various system configurations. For this implementation, a 128-bit data bus was selected as an efficient balance between data throughput and silicon area constraints. This allows for the transfer of 8 bfloat16 data points per memory transaction, which optimizes performance while keeping area overhead manageable. Using a wider bus width could indeed increase the number of processing elements (PEs) that can operate concurrently, but this would also significantly increase the silicon area and power requirements, which are critical factors for resource-constrained edge devices. We designed the accelerator based on these numbers, so we tile the inputs to work on a smaller portion of data. For row and column vectors, one tile consists of 8 data points and is available in one memory transaction. For matrices, one tile includes 8×8 data points and is available in 8 memory transactions. Additionally, the accelerator is designed to connect to a typical single-port SRAM for area efficiency and compatibility with standard memory configurations. With single-port SRAM, simultaneous read and write operations or dual reads within the same clock cycle are not possible.

B. NLU OVERVIEW

NLU supports 20 instructions directly. The instructions are described in Table 1. Figure 4 shows the simplified architecture of NLU with memories and register files. The NLU consists of 3 main units: Dynamic Control Unit (DCU), Tile Allocation Engine (TAE), and Neural Compute Pipeline (NCP).

The Dynamic Control Unit fetches and decodes the instructions, depending on operands and operation type configures the Tile Allocation Unit and Neural Compute Pipeline. Tile Allocation Unit traces the global tile location and operands tile location. The Tile Allocation Unit reads one tile of input operands and stores it in the input pipeline buffer of the Neural Compute Pipeline. The Neural Compute Pipeline executes the operation and stores the results in the output buffer pipeline. Dynamic Control Unit triggers the Tile Allocation Unit to write the results to the appropriate tile of the output operand.

C. TILE ALLOCATION ENGINE (TAE)

In NLU, an operation like elementwise matrix multiplication is divided into several iterations based on the matrix size. One iteration comprises the computations for producing one tile of the output operand. The number of iterations is the same as the number of tiles in the largest array (matrix). The type and number of computations in one iteration are fixed, and only the memory locations for input and output operands may differ.

During each iteration, the NLU reads one tile of data, i.e., 64 values for a matrix and 8 values for a vector, computes the intermediate result, keeps it in the Output Datapath, or stores it in the temporary memory location. Repeat this process for all inputs to calculate the final result of the tile. Then,

TABLE 1. Directly supported instructions in the NLU.

Instruction	Description
E-S-V-A/S/M	Elementwise-Scalar-Vector-Add/Sub/Mul
E-S-M-A/S/M	Elementwise-Scalar-Matrix-Add/Sub/Mul
E-V-V-A/S/M	Elementwise-Vector-Vector-Add/Sub/Mul
Outer-Prod	Outer product
E-V-M-A/S/M	Elementwise-Vector-Matrix-Add/Sub/Mul
E-M-M-A/S/M	Elementwise-Matrix-Matrix-Add/Sub/Mul
V-M-Mul	Vector-Matrix-Multiplication
V-M-MulT	Transposed-Vector-Matrix-Multiplication
ReLU	Rectified Linear Unit
STEP	Step Function

it moves to the next tile. Upon computing the last tile, the computation is completed.

Tile Allocation Engine controls the tile's position, keeps track of the overall tile count, and determines the last tile in a block.

TAE architecture is shown in green in Figure 4. TAE consists of an Adaptive Sequencing Module, MAD (multiply-add unit), Compare, Tile Select, and ADD units. The operand tile location in memory is generated based on several factors, including its array type (row vector, column vector, or matrix), whether it needs to be overread or overwritten, and the tile location. The TAE utilizes the Adaptive Sequencing Module to track the tile location and calculates the current operand address in the data memory.

Tile Allocation Engine manages memory access and optimizes memory usage by reusing temporary memory locations for storing intermediate values.

D. DYNAMIC CONTROL UNIT (DCU)

All information needed to execute one complete scalar, vector, or matrix operation is embedded in the instruction. The instruction memory corresponding to Equation (1), shown in the top part of Figure 4, is 128 bits wide. The instruction includes opcode, input and output operands start address, number of tiles in a column for each operand, overread input operands, overwrite output operand, last instruction, and vector type. The opcode is one of the instructions shown in Table 1. Depending on the operation, certain results are intermediate and relevant only for the current iterations. To reduce extra memory usage and store only one tile for intermediate results, two control signals, namely overread and overwrite, are defined. During a read operation, if overread is set to one, the address of the first tile is chosen in every iteration. Similarly, during a write operation, if overwrite is set to one, NLU only writes to the first tile.

The instruction memory of NLU allows for the definition of multiple blocks, indicated by the *Last* field in Figure 4. In the hardware, all operations within one block are completed before transitioning to the next block. For element-wise

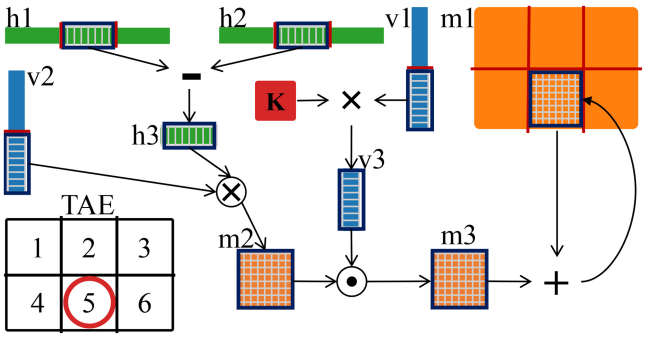


FIGURE 5. Data and operation flow of the NLU.

operations, various computations, as shown in (1), can be grouped within a single block. This is possible because, in each iteration, one tile is computed independently of the others. For training, one or several blocks of instruction memory are used to compute one neural network layer. Vector-matrix multiplication and transpose Vector-matrix multiplication, however, must be executed in separate layers. For these operations, all values are computed initially before moving on to the next operation (layer). This sequential approach is necessary, as each iteration depends on the previous one, and the final result becomes available after the last iteration.

The Dynamic Control Unit architecture is shown in orange in Figure 4 and consists of Fetch and Decode Units, Address, Operation, Buffer, Operand, and Memory controllers. Depending on the incoming instructions, the DCU dynamically schedules the Tile Allocation Engine and the Neural Compute Pipeline to perform complicated vector or matrix operations.

The DCU fetches and decodes instructions. Based on the opcode and operands type and addresses, the DCU sends signals to the Tile Allocation Engine to read one tile of each operand from the memory, storing them in the input pipeline buffer of the Neural Compute Pipeline. The DCU configures the Neural Compute Pipeline for the computation. The execution result is written to the output pipeline buffer. Finally, the DCU sends a signal to the Tile Allocation Engine to read data from the buffer and store it in the memory.

E. NEURAL COMPUTE PIPELINE (NCP)

The Neural Compute Pipeline utilizes an input pipeline buffer, arithmetic logic unit (ALU), and output pipeline buffer and can execute complex operations on scalar, vector, and matrix operands.

The input pipeline buffer architecture is depicted in blue in Figure 4. Input operands are read from the data memory one by one, and they are buffered before being sent to the ALU. The input pipeline buffer consists of Buffer Manager, Expand, and Reorder units. The Buffer Manager is responsible for buffering the input data for the ALU. For element-wise column vector-matrix operations and outer product, the Column Vector Decoder is utilized. In these

scenarios, each vector element interacts with eight elements of a matrix or vector. The Expand unit takes a 16-bit input and replicates it eight times to produce a 128-bit output. This is employed when the input is a scalar, and the data is encoded in the instruction rather than a data memory location. The Reorder unit distributes the input data into two buffers, allowing them to be accumulated by the ALU.

As shown in red in Figure 4, the arithmetic logic unit (ALU) comprises eight bfloat16 fused multiply-add (FMA) units. It is also capable of performing addition, subtraction, and multiplication. The ALU multiplies the first operand with the second operand and adds it to the third operand. The Bfloat16 FMA is specifically designed for deep learning applications, optimizing power consumption and area efficiency. To achieve this, values in the subnormal range and underflows are treated as zero. Additionally, the sparse detection unit evaluates the exponent part of the operands and either skips the operation or forwards it when one of the operands is zero.

The FMA unit supports two rounding modes: nearest neighbor and stochastic rounding. Stochastic rounding is robust to the quality of the random number generator [59]; in this case, a 10-bit Linear Feedback Shift Register (LFSR) was used to generate 8-bit pseudo-random numbers (PRNG), which are then applied in the rounding stage of the FMA unit to determine the rounding bit randomly. The rounding mode is controlled by the register file, which must be configured before the accelerator is run. Eight FMA units in ALU work in parallel and execute eight bfloat16 data in every clock cycle in a pipeline. The ALU can perform various scalar, vector, and matrix operations by controlling inputs and operation types.

The output pipeline buffer architecture is depicted in yellow in Figure 4, it stores the result of operations for one tile before data is written to the memory or forwarded to the input pipeline buffer. The output pipeline buffer improves performance and energy efficiency by reducing unnecessary memory access.

The efficient pipeline buffer design, which can store one tile of input and output operands, minimizes the need for data transfers, improves data reuse, reduces energy consumption, and accelerates processing.

F. DATA AND OPERATION FLOW

Figure 5 shows how the operations in (1) are executed in NLU to compute the one tile of matrix $m1$. Dynamic Control Unit (DCU) fetches the instructions and configures the Tile Allocation Engine (TAE) and Neural Compute Pipeline (NCP). Based on the global tile number (in Figure 5 global tile number is 5), TAE loads the corresponding tile of input operands into the input pipeline buffer of the NCP. NCP executes the operations and produces the final result. Finally, TAE copies one tile of the result to the memory.

In summary, in this section, we introduced the architectural details of our Neural Learning Unit (NLU) and described how the Dynamic Control Unit (DCU), Tile Allocation Engine (TAE), and Neural Compute Pipeline (NCP) work together to support various deep learning training operations. Building on this hardware design, the next section describes our implementation process and presents in-depth measurements of the system's performance, power consumption, and area efficiency.

V. IMPLEMENTATION AND MEASUREMENTS

Building on the hardware design presented in Section IV, we now describe the practical realization of our Neural Learning Unit (NLU) within a system-on-chip (SoC) prototype. Specifically, this section details our chip fabrication process, outlines the measurement setup and evaluation methodology, and reports on the area, power, and performance metrics of the implemented design. By examining these measurements, we demonstrate the effectiveness of our approach and validate the feasibility of on-device training with the NLU.

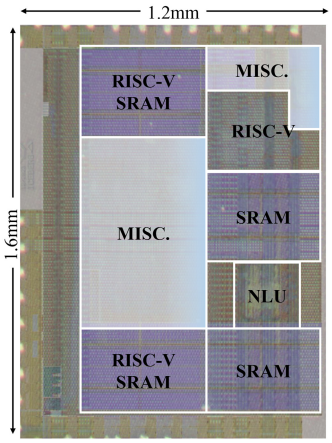
A. SYSTEM ON CHIP (SOC)

To evaluate NLU's key performance indicators (KPIs), we developed a System on Chip (SoC) featuring NLU, RISC-V, and memory components and fabricated a chip using GlobalFoundries 22nm FDSOI technology. Figure 6 shows the chip photograph and specifications. NLU achieves 367 MFLOPS for vector-matrix multiplication while consuming an average peak power of 0.379 mW across four boards. The high-level architecture of the SoC is illustrated in Figure 7(a). This SoC comprises two distinct domains: the accelerator domain contains NLU and two 64KB memories with 128-bit data buses, operating at 40MHz, and the processor domain contains a 32-bit RISC-V processor [60] equipped with a 32-bit floating-point unit (FPU) [61] (RV32IMFC ISA), along with two 64KB memory, operating at 20MHz.

The accelerator connects to the processor via the Open Bus Interface (OBI) protocol and uses its own memory-mapped addresses. While the accelerator's primary bus protocol is OBI, the design can be extended to AMBA-based systems with minimal modifications.

First, the main processor configures the NLU and initiates it. Once all operations have been performed on the data, the NLU sets a *Finished* flag to 1 and sends an interrupt to the processor, signaling that the computations have been completed.

As discussed in Section IV-D of Section IV, one or more instructions can be grouped into a single block. At startup, the main processor programs the NLU's instruction memory, which consists of multiple instruction blocks. The processor maintains the start address of each block and initiates execution by writing to the NLU's program counter. The overhead of switching between blocks is negligible, as only the program counter of the NLU is modified. Once a block has been executed, the NLU sets the *Finished* flag to 1 and sends an interrupt to the processor, signaling the completion of computations.



Category	Specification
Technology	22 nm
Suply Voltage	0.55 V
Die Area	1.92 mm^2
RISC-V	
Max. Frequency	20 MHz
SRAM	128 KB
Data Type	FP32
NLU	
Max. Frequency	40 MHz
SRAM	128 KB
Data Type	BF16
Peak Performance*	367 MFLOPS
Energy Efficiency*	968 GFLOPS/W
Power*	0.379 mW

* Based on vector-matrix multiplication

FIGURE 6. Chip micrograph and summary.

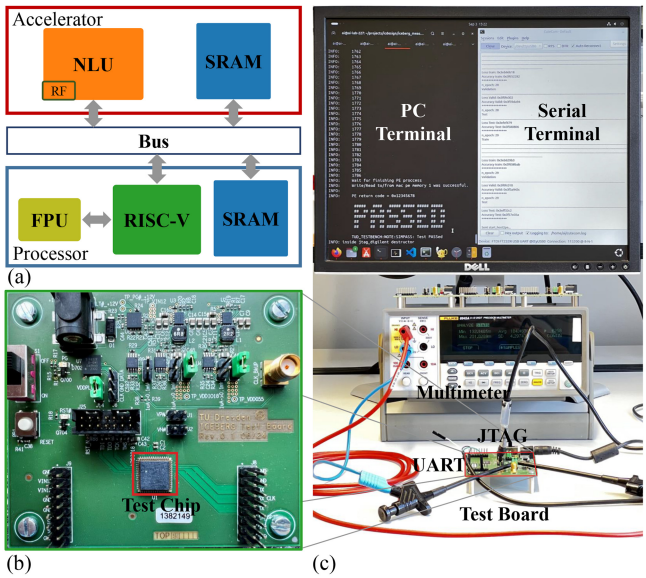


FIGURE 7. (a) The architecture of the chip. (b) Test board. (c) Lab setup.

Subsequently, we designed a PCB for the chip and conducted measurements in the laboratory, as depicted in Figure 7(b) and Figure 7(c).

B. RESULTS AND ANALYSIS: SINGLE OPERATIONS

Four KPIs are typically considered when evaluating hardware for training deep learning applications: programmability, power, performance, and area (PPPA). In the upcoming sections, we will explore these KPIs in detail.

1) PROGRAMMABILITY

Programmability is the most critical KPI in deep learning hardware. It determines the complexity or simplicity of configuring the accelerator for different operations. This factor also influences the complexity of designing a compiler. One example of poor programmability is the use of systolic arrays for tasks other than matrix multiplication, such as elementwise multiplication. Designing a compiler to reorder the matrix for executing elementwise multiplication is a complex task.

Algorithm 1 Configuring NLU Instruction Memory to Perform Vector-Matrix Multiplication

```
inst.bits.opcode        = OPCODE_VMMac;
inst.bits.addr_A        = vec_A_str_addr;
inst.bits.addr_B        = mat_B_str_addr;
inst.bits.addr_R        = vec_R_str_addr;
inst.bits.numColTile_A = 1;
inst.bits.numColTile_B = 6;
inst.bits.numColTile_R = 6;
inst.bits.OR_A          = 0;
inst.bits.OR_B          = 0;
inst.bits.OW_R          = 0;
inst.bits.Last          = 1;
inst.bits.VecType       = Col;

for (i=0; i<4; i++)
    apb.write(sram_addr+i*4, 0XF, inst.byte[i]);
```

To configure NLU in C, we defined a union containing the bit and byte definitions of the instruction memory. We then filled the bit fields in the union and wrote the byte fields to the instruction memory. An example of vector-matrix multiplication for a vector of 72x1 and a matrix of 72x48 is illustrated in Algorithm 1. As evident, a compiler can effortlessly generate configuration code for NLU.

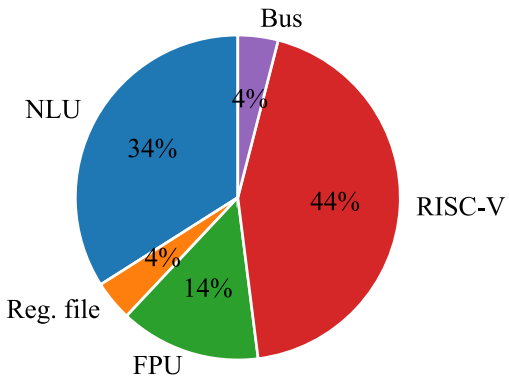


FIGURE 8. Area breakdown of SoC without considering the memory.

2) AREA

The core area of the SoC with memories is 0.95 mm^2, of which 52% is occupied by memory. The area breakdown of SoC without considering the memory is shown in Figure 8. RISC-V takes 20046 μm^2, FPU 6422 μm^2, NLU 15433 μm^2, register file 1773 μm^2, and bus logic 1,863 μm^2. In NLU, the ALU makes up 7882 μm^2, and the remaining area is used for control logic. Despite its more complicated functionality, the NLU is only 2.4 times larger than the FPU and also 23% smaller than the RISC-V.

TABLE 2. Performance comparison of training execution between RISC-V and RISC-V with NLU.

Category / Algorithm		MLP	GRU	SRNN
Execution [Cycles]	RISC-V	162,150	559,852	188,770
	RISC-V + NLU	8,329	22,963	14,592
Power [mW]	RISC-V	0.83	0.82	0.83
	RISC-V + NLU	1.17	1.07	1.12
Speedup [Cycles]		19.47X	24.38X	12.94X
Speedup [Wall Clock]		38.94X	48.76X	25.88X
Energy Efficiency		27.62	37.37	19.18

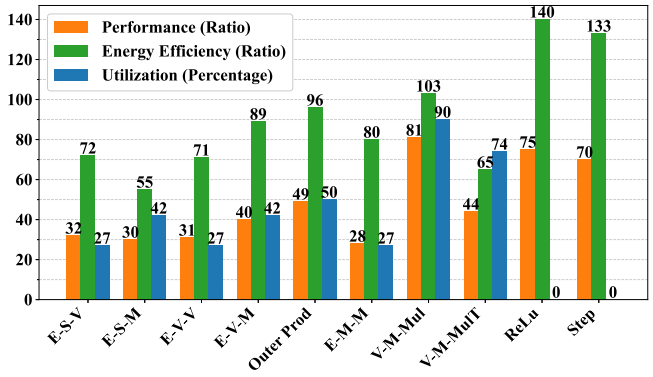


FIGURE 9. Comparison of the performance and energy efficiency of the NLU versus the RISC-V implementation across different operations, based on cycle-accurate simulations and post-layout power analysis. "Elem" stands for elementwise, "S" for scalar, "V" for vector, and "M" for matrix.

3) PERFORMANCE AND ENERGY

We utilized a cycle-accurate simulator to evaluate NLU's performance. For individual operations, we measured the number of cycles it takes for the RISC-V with an FPU and NLU to execute the operation. The results are summarized in Figure 9. The vector size is 72, and the matrix size is 72 × 72 for all evaluations. The orange bar illustrates the speedup of NLU relative to RISC-V, with NLU capable of executing single operations up to 81 times faster than RISC-V. This speedup is achieved through data reuse in computation. For example, in a vector-matrix operation, 8 vector elements and 64 matrix elements are loaded from memory, and the vector is multiplied by the matrix and accumulated with the previous result. This approach allows 512 multiply-and-accumulate operations to be performed before the data is written back to memory. In contrast, RISC-V reads data from memory, performs one operation, and writes the result back to memory, with most clock cycles consumed by loop overhead and memory transactions. The green bar represents the energy consumption of the RISC-V processor relative to the NLU, based on post-layout power simulations. The blue bar demonstrates the FMA utilization during execution, indicating the percentage of clock cycles spent on execution versus on memory transactions.

A direct relationship exists between performance and FMA utilization. For elementwise operations, most of the execution time is spent on memory transactions, while for vector-matrix multiplication, most of the execution time is spent on computation. This is because, in vector-matrix multiplication, NLU can reuse one tile of data several times before writing it back to memory. For ReLU and STEP activation functions, the ALU is not used, resulting in zero FMA utilization. Additionally, vector-matrix multiplication is executed faster than transpose vector-matrix multiplication because, as described earlier, the computation order in vector-matrix multiplication matches the NLU addressing mode, resulting in fewer memory accesses.

Compared to a RISC-V Core with 8 FPUs, NLU occupies 15433 μm^2, while a RISC-V core with 8 FPUs requires 71422 μm^2 of silicon. In other words, a RISC-V core with 8 FPUs is 4.6× larger than a single NLU. Even when accounting for the fact that a RISC-V core with 8 FPUs is 8× faster than a RISC-V core with only one FPU, NLU remains up to 10.125× faster and is also more power-efficient.

C. RESULTS AND ANALYSIS: ALGORITHMS

To evaluate the performance of NLU on deep learning algorithms, we consider the same algorithms analyzed in Section II-C.

For the MLP, we use a two-layer fully connected network with ReLU and SoftMax activation functions, including the backward pass. All operations except SoftMax can be executed on NLU.

For the GRU, we implemented one layer network including the forward and backward pass using backpropagation through time (BPTT). All operations except the sigmoid and tanh activation functions can be accelerated with NLU.

For the SRNN, we implemented a three-layer network on the chip, where the hidden layer contains spiking neurons. We used the e-prop learning rule for training this network. In this case, all operations can be accelerated using NLU.

We used a cycle-accurate simulator to measure number of clock cycles for the MLP, GRU, and e-prop algorithms using a pure RISC-V implementation with an FPU and compared these results to a combined software and hardware implementation.

The MLP, GRU, and e-prop algorithms were implemented on the chip using RISC-V, both with and without NLU. We employed a cycle-accurate simulator to count clock cycles and used a lab setup to measure the power consumption for each case. The results are presented in Table 2. All performance metrics reported in Table 2, including clock cycles, power consumption, and speedup, take into account the data transmission overhead between the RISC-V core and the Neural Learning Unit (NLU). This ensures an accurate representation of the system's performance. With NLU, the training process runs up to 48.76× faster and is 37.37× more energy-efficient.

TABLE 3. Comparison with deep learning training chips.

Category	Vecim [62]	RedMulE [33]	DF-LNPU [36]	Trainer [38]	OCT [40]	CTP [39]	This work
Technology [nm]	65	22	65	28	28	28	22
Type	Vector Coprocessor	GEMM	CNN, FC	CNN, FC	CNN, FC	CNN	General
Voltage [V]	1	0.65 - 0.8	0.7 - 1.1	0.56 - 1.0	0.48	0.6 - 1.1	0.55
Frequency [MHz]	250	470 - 630	25 - 200	40 - 440	20-200	75-340	40
On-chip SRAM	32 Kb	128 KB	329 KB	634 KB	64 KB	1.25 MB	256 KB
Data format	BF16, FP16	FP16	FP8, FP16	FP8, FP16	INT8	FP8, FP16	BF16
Power [mW]	110	59.3 - 116	38.1 - 414	23 - 363	0.836 - 18	623.7	0.379
Peak Performance [GFLOPS]	25.3	44.8 - 58.5	19.4 - 155.2	450 - 900	38.4	560	0.367
Area [mm^2]	4	0.64	16	20.96	2	16.4	0.015
Energy Efficiency [GFLOPS/W]	230	775 - 506	509 - 375	2140 - 4280	2130 - 4690	898	968
Coprocessor	RISC-V	On-Chip RISC-V	FPGA	FPGA	FPGA	FPGA	On-Chip RISC-V

D. ON-DEVICE TRAINING

We utilized the Google Speech Commands (GSC) dataset with 8 categories for on-device learning. The dataset was preprocessed using Mel-Frequency Cepstral Coefficients (MFCCs). The preprocessing step took one-second audio files as input and produced a 32 (input) by 80 (time step) array for each file. For training, we randomly selected only 40% of the dataset to reduce training time and ensure it could be completed within the experiment session. The dataset contains 9,781 samples, with 64% used for the training set, 18% for the validation set, and 18% for the test set. To reduce communication time between the host and chip, we compressed the data to float8 format before sending it to the chip. The total dataset size is 24 MB.

On the chip, we implemented a three-layer SRNN network with 32 input, 56 hidden, and 8 output neurons. The input and output layers are dense, while the hidden layer is spiking recurrent. We applied the e-prop learning rule [54], [55] to train the network. Compared to backpropagation through time (BPTT), e-prop is a memory-efficient algorithm and is well-suited for our lab setup. We employed a mixed-precision training approach. Figure 10 illustrates the training process within the chip.

The loss function (cross-entropy loss) and the error function (softmax), which require natural logarithmic calculations, are executed in RISC-V. Additionally, the ADAM optimization algorithm, which involves division and requires greater precision, is also executed in RISC-V. Forward pass and gradient computation, which involves vector and matrix operations and ReLU and STEP functions, are performed entirely on the NLU.

To demonstrate the real-world functionality of on-chip learning, we trained the model from scratch using five different sets of randomly initialized weights on the GSC dataset. For each set of weights, we ran the training process and computed the accuracy and loss for the training, validation, and test datasets directly on the chip. The average accuracy for the training and validation sets over five different runs, across 15 epochs, using round-to-nearest-even rounding, is shown in Figure 11. It achieved a maximum training accuracy of 89.18%, a best validation accuracy of 84.93%, and a test accuracy of 84.51%, with the best validation accuracy occurring at an average of 8.6 epochs.

We conducted the same test using stochastic rounding, achieving a maximum training accuracy of 89.34%, a best validation accuracy of 85.25%, and a test accuracy of 84.38%, with the best validation accuracy occurring at an average of 8 epochs. Compared to round-to-nearest-even rounding, stochastic rounding achieved slightly better accuracy on the training and validation sets but slightly worse accuracy on the test set.

VI. COMPARING WITH STATE-OF-THE-ART

Table 3 compares NLU with different state-of-the-art deep-learning training chips. For a fair comparison, we consider only dense computations. The NLU achieves a peak performance of 367 MFLOPS when computing vector-matrix multiplication and consumes a peak power of 379 mW.

We compared our work with Vecim [62], a compute-in-memory (CIM) vector processor that supports in-memory 16-bit floating-point multiplication and addition, significantly reducing data transfers between the vector register file and ALU, and is compatible with the RISC-V instruction set architecture (ISA). The peak performance of Vecim is 68.9× higher than NLU; however, it consumes 290.2× more power and occupies 266.7× more silicon area.

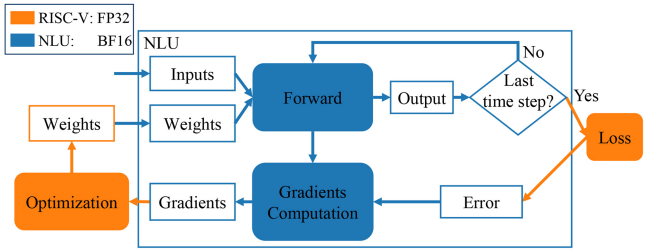


FIGURE 10. On-device training process. The RISC-V provides input data, NLU performs the forward pass and predicts the output. After the last time step, the RISC-V computes loss and error (the difference between predicted and ground truth output). NLU then uses this information to compute the gradients. Finally, the RISC-V applies ADAM optimization to update the weights.

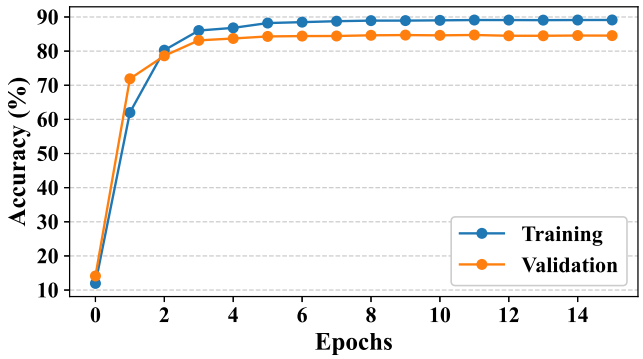


FIGURE 11. On-device learning results for training and validation set with round-to-nearest-even rounding usind GSC dataset (average results over 5 runs).

RedMule [33] is a TinyML training accelerator that supports GEMM-Ops (matrix multiplication, addition, maximum, and minimum) that achieves up to 159 times higher performance than NLU, but consumes 214.8 times more power and is 42.7 times larger than NLU. Domain-specific accelerators compared to NLU support a limited number of operations and heavily rely on the processor in the training process and actual TFLOPS in real word applications is less than these numbers.

Furthermore, we evaluated our work in comparison with network-specific accelerators. DF-LNPU [36] is a learning processor that uses direct feedback alignment (DFA) [63] learning rule to train convolutional neural networks (CNN), and fully connected networks (FC). DFA [63] has limited use cases and only works for shallow networks and small datasets. DF-LNPU is 422× higher performance compared to NLU but consumes 1092× more power and is 1067 times larger than NLU. In addition, DF-LNPU can only utilize DFA learning rule and unlike NLU is not flexible.

Trainer [38] is another network-specific accelerator that is optimized to train convolutional neural networks. Compared to NLU, it is 2452× higher performance and consumes 958× more power.

OCT [40] is an 8-bit fixed-point network-specific accelerator that is optimized to train convolutional neural networks. Due to its fixed-point support, it has low power consumption and also very limited application.

CTP [39] is another network-specific accelerator that is optimized to train convolutional neural networks. CTP has 1526× higher performance and consumes 1646× more power and 1093× more area compared to the NLU.

Compared to state-of-the-art training processors, NLU consumes less power, occupies a smaller area, and is highly configurable which makes it a good fit for edge or embedded training applications.

VII. CONCLUSION

This paper introduced the Neural Learning Unit (NLU), a highly configurable and flexible on-device learning accelerator. By supporting a wide range of operations–including elementwise arithmetic, vector-matrix multiplication, and nonlinear activation functions–the NLU significantly reduces both computation time and energy consumption compared to a baseline RISC-V with FPU implementation. Integrating the NLU into a system-on-chip (SoC) with a RISC-V core demonstrated up to 24.38× faster training performance and up to 37.37× energy savings. Moreover, we validated its real-world capability by successfully training a three-layer recurrent neural network from scratch on the Google Speech Commands dataset, achieving 84.51% accuracy on the test set.

Although our results are promising, several opportunities exist to further optimize and expand the NLU's capabilities. First, increasing the range of supported operations (for instance, additional activation functions or specialized layers) could expand its applicability to more advanced architectures. Second, adapting the accelerator for larger buffer sizes or higher internal bandwidth may yield additional gains in throughput. Third, exploring a design with multiple NLUs interconnected via a network-on-chip (NoC) could improve scalability and meet the demands of larger or more complex neural networks. These efforts will help maximize the advantages of the proposed accelerator and address any limitations identified in the current design, ultimately creating opportunities for more efficient on-device training of deep neural networks.

REFERENCES

[1] J. Lee, S. Kang, J. Lee, D. Shin, D. Han, and H.-J. Yoo, "The hardware and algorithm co-design for energy-efficient DNN processor on edge/mobile devices," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 10, pp. 3458–3470, Oct. 2020.

[2] H. Askr, E. Elgeldawi, H. A. Ella, Y. A. M. M. Elshaier, M. M. Gomaa, and A. E. Hassanien, "Deep learning in drug discovery: An integrative review and future challenges," *Artif. Intell. Rev.*, vol. 56, no. 7, pp. 5975–6037, Jul. 2023.

[3] R. Zheng, L. Qu, B. Cui, Y. Shi, and H. Yin, "AutoML for deep recommender systems: A survey," *ACM Trans. Inf. Syst.*, vol. 41, no. 4, pp. 1–38, Mar. 2023.

[4] N. Youneszade, M. Marjani, and C. P. Pei, "Deep learning in cervical cancer diagnosis: Architecture, opportunities, and open research challenges," *IEEE Access*, vol. 11, pp. 6133–6149, 2023.

[5] J. Bharadiya, "A comprehensive survey of deep learning techniques natural language processing," *Eur. J. Technol.*, vol. 7, no. 1, pp. 58–66, May 2023.

[6] A. N. Mazumder et al., "A survey on the optimization of neural network accelerators for micro-AI on-device inference," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 11, no. 4, pp. 532–547, Dec. 2021.

[7] K. Rungsuptaweekoon, V. Visoottiviseth, and R. Takano, "Evaluating the power efficiency of deep learning inference on embedded GPU systems," in *Proc. 2nd Int. Conf. Inf. Technol. (INCIT)*, 2017, pp. 1–5.

[8] M. Arafa et al., "Cascade lake: Next generation Intel Xeon scalable processor," *IEEE Micro*, vol. 39, no. 2, pp. 29–36, Mar./Apr. 2019.

[9] J. Choquette and W. Gandhi, "NVIDIA A100 GPU: Performance & Innovation for GPU Computing," in *Proc. IEEE Hot Chips 32 Symp. (HCS)*, Aug. 2020, pp. 1–43.

[10] L. Kljucaric and A. D. George, "Deep learning inferencing with high-performance hardware accelerators," *ACM Trans. Intell. Syst. Technol.*, vol. 14, no. 4, pp. 68, pp. 1–25, Jun. 2023.

[11] C. A. Künas, M. S. Serpa, and P. O. A. Navaux, "Exploiting hardware accelerators in clouds," in *High Performance Computing in Clouds: Moving HPC Applications to a Scalable and Cost-Effective Environment*, E. Borin, L. M. A. Drummond, J.-L. Gaudiot, A. Melo, M. M. Alves, and P. O. A. Navaux, Eds., Cham, Switzerland: Springer, 2023, pp. 127–144.

[12] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, and S. Han, "Tiny machine learning: Progress and futures [feature]," *IEEE Circuits Syst. Mag.*, vol. 23, no. 3, pp. 8–34, Oct. 2023.

[13] V. Sze, Y.-H. Chen, T.-J. 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.

[14] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proc. IEEE*, vol. 107, no. 8, pp. 1655–1674, Aug. 2019.

[15] H. Cai, C. Gan, L. Zhu, and S. Han, "TinyTL: Reduce memory, not parameters for efficient on-device learning," in *Advances in Neural Information Processing Systems*, vol. 33. Red Hook, NY, USA: Curran Assoc., Inc., 2020, pp. 11285–11297.

[16] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wireless Commun.*, vol. 19, no. 1, pp. 447–457, Jan. 2020.

[17] J. Lee and H.-J. Yoo, "An overview of energy-efficient hardware accelerators for on-device deep-neural-network training," *IEEE Open J. Solid-State Circuits Soc.*, vol. 1, pp. 115–128, 2021.

[18] R. Singh and S. S. Gill, "Edge AI: A survey," *Internet Things Cyber-Phys. Syst.*, vol. 3, pp. 71–92, Mar. 2023.

[19] A. Ancilotto, F. Paissan, and E. Farella, "XiNet: Efficient neural networks for TinyML," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2023, pp. 16968–16977.

[20] K. Xu, H. Zhang, Y. Li, Y. Zhang, R. Lai, and Y. Liu, "An ultra-low power TinyML system for real-time visual processing at edge," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 70, no. 7, pp. 2640–2644, Jul. 2023.

[21] V. Jain, S. Giraldo, J. D. Roose, L. Mei, B. Boons, and M. Verhelst, "TinyVers: A tiny versatile system-on-chip with state-retentive eMRAM for ML inference at the extreme edge," *IEEE J. Solid-State Circuits*, vol. 58, no. 8, pp. 2360–2371, Aug. 2023.

[22] P. Behnam et al., "Hardware–software co-design for real-time latency–accuracy navigation in tiny machine learning applications," *IEEE Micro*, vol. 43, no. 6, pp. 93–101, Nov./Dec. 2023.

[23] S. Tuli and N. K. Jha, "AccelTran: A sparsity-aware accelerator for dynamic inference with transformers," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 11, pp. 4038–4051, Nov. 2023.

[24] G. A. Lewis, "Characterizing and detecting mismatch and predicting inference degradation in ML systems," Dept. Softw. Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA, USA, Rep. DM21-0059, 2021. https://apps.dtic.mil/sti/citations/AD1146933

[25] F. Bayram, B. S. Ahmed, and A. Kassler, "From concept drift to model degradation: An overview on performance-aware drift detectors," *Knowl. Based Syst.*, vol. 245, Jun. 2022, Art. no. 108632.

[26] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Int. Conf. Artif. Intell. Statist.*, Apr. 2017, pp. 1273–1282.

[27] J. Lu, J. Huang, and Z. Wang, "THETA: A high-efficiency training accelerator for DNNs with triple-side sparsity exploration," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 8, pp. 1034–1046, Aug. 2022.

[28] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference workloads for modern deep learning methods," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10.

[29] S. Shukla et al., "A scalable multi-TeraOPS core for AI training and inference," *IEEE Solid-State Circuits Let.*, vol. 1, no. 12, pp. 217–220, Dec. 2018.

[30] E. Qin et al., "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 58–70.

[31] G. Jeong et al., "VEGETA: Vertically-integrated extensions for sparse/dense GEMM tile acceleration on CPUs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb. 2023, pp. 259–272.

[32] A. Samajdar, E. Qin, M. Pellauer, and T. Krishna, "Self adaptive reconfigurable arrays (SARA): Learning flexible GEMM accelerator configuration and mapping-space using ML," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, New York, NY, USA, Aug. 2022, pp. 583–588.

[33] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, "RedMule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and TinyML training acceleration," *Future Gener. Comput. Syst.*, vol. 149, pp. 122–135, Dec. 2023.

[34] N. Jouppi et al., "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proc. 50th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, Jun. 2023, pp. 1–14.

[35] R. Das and T. Krishna, *DNN Accelerator Architecture–SIMD or Systolic? Computer Architecture Today*, ACM SIGARCH, New York, NY, USA, 2018.

[36] D. Han, J. Lee, and H.-J. Yoo, "DF-LNPU: A pipelined direct feedback alignment-based deep neural network learning processor for fast online learning," *IEEE J. Solid-State Circuits*, vol. 56, no. 5, pp. 1630–1640, May 2021.

[37] C. Frenkel and G. Indiveri, "ReckOn: A 28nm sub-mm2 task-agnostic spiking recurrent neural network processor enabling on-chip learning over second-long timescales," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2022, pp. 1–3.

[38] Y. Wang et al., "Trainer: An energy-efficient edge-device training processor supporting dynamic weight pruning," *IEEE J. Solid-State Circuits*, vol. 57, no. 10, pp. 3164–3178, Oct. 2022.

[39] S. K. Venkataramanaiah et al., "A 28-nm 8-bit floating-point tensor core-based programmable CNN training processor with dynamic structured sparsity," *IEEE J. Solid-State Circuits*, vol. 58, no. 7, pp. 1885–1897, Jul. 2023.

[40] J. Qian, H. Ge, Y. Lu, and W. Shan, "A 4.69-TOPS/W training, 2.34-μ J/image inference on-chip training accelerator with inference-compatible backpropagation and design space exploration in 28-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 60, no. 1, pp. 298–307, Jan. 2025.

[41] B. Li et al., "DQ-STP: An efficient sparse on-device training processor based on low-rank decomposition and quantization for DNN," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 71, no. 4, pp. 1665–1678, Apr. 2024.

[42] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, New York, NY, USA, Apr. 2021, pp. 687–701.

[43] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, Jun. 2017, pp. 1–12.

[44] T. Norrie et al., "The design process for Google's training chips: TPUv2 and TPUv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, Mar./Apr. 2021.

[45] S. K. Lee et al., "A 7-nm four-core mixed-precision ai chip with 26.2-TFLOPS hybrid-FP8 training, 104.9-TOPS INT4 inference, and workload-aware throttling," *IEEE J. Solid-State Circuits*, vol. 57, no. 1, pp. 182–197, Jan. 2022.

[46] R. Prabhakar, S. Jairath, and J. L. Shin, "SambaNova SN10 RDU: A 7nm dataflow architecture to accelerate software 2.0," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2022, pp. 350–352.

[47] J. Vasiljevic et al., "Compute substrate for software 2.0," *IEEE Micro*, vol. 41, no. 2, pp. 50–55, Mar./Apr. 2021.

[48] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.

[49] P. Micikevicius et al., "Mixed precision training," in *Proc. Int. Conf. Learn. Represent.*, Feb. 2018, pp. 1–14.

[50] D. Kalamkar et al., "A study of BFLOAT16 for deep learning training," 2019, *arXiv:1905.12322*.
[51] A. Agrawal et al., "DLFloat: A 16-b floating point format designed for deep learning training and inference," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Jun. 2019, pp. 92–95.
[52] "Deep learning systems." Accessed: Nov. 19, 2024. [Online]. Available: https://dlsyscourse.org/
[53] "10.2. Gated recurrent units (GRU)—Dive into deep learning 1.0.3 documentation." [Online]. Available: https://d2l.ai/chapter_recurrent-modern/gru.html
[54] G. Bellec et al., "A solution to the learning dilemma for recurrent networks of spiking neurons," *Nat. Commun.*, vol. 11, no. 1, p. 3625, Jul. 2020.
[55] A. Rostami, B. Vogginger, Y. Yan, and C. G. Mayr, "E-prop on SpiNNaker 2: Exploring online learning in spiking RNNs on neuromorphic hardware," *Front. Neurosci.*, vol. 16, Nov. 2022, Art. no. 1018006.
[56] Q. Chen, Z. Zhuo, and W. Wang, "Bert for joint intent classification and slot filling," 2019, *arXiv:1902.10909*.
[57] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. 10th Int. Workshop Front. Handwriting Recognit.*, 2006, pp. 1–7.
[58] S. Chetlur et al., "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[59] S.-E. Chang et al., "ESRU: Extremely low-bit and hardware-efficient stochastic rounding unit design for low-bit DNN training," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2023, pp. 1–6.
[60] M. Gautschi et al., "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017.
[61] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 4, pp. 774–787, Apr. 2021.
[62] Y. Wang, M. Yang, C.-P. Lo, and J. P. Kulkarni, "30.6 Vecim: A 289.13GOPS/W RISC-V vector co-processor with compute-in-memory vector register file for efficient high-performance computing," in *Proc. IEEE In. Solid-State Circuits Conf. (ISSCC)*, 2024, pp. 492–494.
[63] A. Nøkland, "Direct feedback alignment provides learning in deep neural networks," in *Advances in Neural Information Processing Systems*, vol. 29. Red Hook, NY, USA: Curran Assoc., Inc., 2016.