Accelerating Deep Learning Inference in Constrained Embedded Devices Using Hardware Loops and a Dot Product Unit

JURE VREČA 1, KARL J. X. STURM 2, (Student Member, IEEE), ERNEST GUNGL 1, (Member, IEEE), FARHAD MERCHANT 2, PAOLO BIENTINESI 3, RAINER LEUPERS 2, AND ZMAGO BREZOČNIK 1, (Member, IEEE)

1Faculty of Electrical Engineering and Computer Science, University of Maribor, 2000 Maribor, Slovenia
2Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, 52056 Aachen, Germany
3Department of Computing Science, Umeå University, 901 87 Umeå, Sweden

Corresponding author: Jure Vreča (jure.vreca@student.um.si)

ABSTRACT

Deep learning algorithms have seen success in a wide variety of applications, such as machine translation, image and speech recognition, and self-driving cars. However, these algorithms have only recently gained a foothold in the embedded systems domain. Most embedded systems are based on cheap microcontrollers with limited memory capacity, and, thus, are typically seen as not capable of running deep learning algorithms. Nevertheless, we consider that advancements in compression of neural networks and neural network architecture, coupled with an optimized instruction set architecture, could make microcontroller-grade processors suitable for specific low-intensity deep learning applications. We propose a simple instruction set extension with two main components—hardware loops and dot product instructions. To evaluate the effectiveness of the extension, we developed optimized assembly functions for the fully connected and convolutional neural network layers. When using the extensions and the optimized assembly functions, we achieve an average clock cycle count decrease of 73% for a small scale convolutional neural network. On a per layer base, our optimizations decrease the clock cycle count for fully connected layers and convolutional layers by 72% and 78%, respectively. The average energy consumption per inference decreases by 73%. We have shown that adding just hardware loops and dot product instructions has a significant positive effect on processor efficiency in computing neural network functions.

INDEX TERMS

Deep learning, embedded systems, instruction set optimization, RISC-V.

I. INTRODUCTION

Typically, deep learning algorithms are reserved for powerful general-purpose processors, because the convolutional neural networks routinely have millions of parameters. AlexNet [1], for example, has around 60 million parameters. Such complexity is far too much for memory-constrained microcontrollers that have memory sizes specified in kilobytes. There are, however, many cases where deep learning algorithms could improve the functionality of embedded systems [2]. For example, in [3], an early seizure detection system is proposed, based on a convolutional neural network running on a microcontroller implanted in the body. The system measures electroencephalography (EEG) data and feeds it to the neural network, which determines the seizure activity. They implemented the neural network on a low power microcontroller from Texas Instruments. Some embedded system designers work around the dilemma of limited resources by processing neural networks in the cloud [4]. However, this solution is limited to areas with access to the Internet. Cloud processing also has other disadvantages, such as privacy concerns, security, high latency, communication power consumption, and reliability. Embedded systems are mostly built around microcontrollers, because they are inexpensive and easy to use. Recent advancements in compression of neural networks [5], [6] and...
advanced neural network architecture [7], [8] have opened new possibilities. We believe that combining these advances with a limited instruction set extension could provide the ability to run low-intensity deep learning applications on low-cost microcontrollers. The extensions must be a good compromise between performance and the hardware area increase of the microcontroller.

Deep learning algorithms perform massive arithmetic computations. To speed up these algorithms at a reasonable price in hardware, we propose an instruction set extension comprised of two instruction types—hardware loops and dot product instructions. Hardware loops, also known as zero-overhead loops, lower the overhead of branch instructions in small body loops, and dot product instructions accelerate arithmetic computation.

The main contributions of this article are as follows:

- we propose an approach for computing neural network functions that are optimized for the use of hardware loops and dot product instructions,
- we evaluate the effectiveness of hardware loops and dot product instructions for performing deep learning functions, and
- we achieved a reduction in the dynamic instruction count, an average clock cycle count, and an average energy consumption of 66%, 73%, and 73%, respectively.

Deep learning algorithms are used increasingly in smart applications. Some of them also run in Internet of Things (IoT) devices. IoT Analytics reports that, by 2025, the number of IoT devices will rise to 22 billion [9]. The motivation for our work stems from the fact that the rise of the IoT will increase the need for low-cost devices built around a single microcontroller capable of supporting deep learning algorithms. Accelerating deep learning inference in constrained embedded devices, presented in this article, is our attempt in this direction.

The rest of this article is organized as follows. Section II presents the related work in hardware and software improvements aimed at speeding up neural network computation. Section III introduces the RISCY core briefly, and discusses hardware loops and the dot product extensions. Section IV shows our experimental setup. Section V first presents a simple neural network that we have developed and ported to our system. It then shows how we have optimized our software for the particular neural network layers. The empirically obtained results are presented and discussed in Section VI. Finally, Section VII contains the conclusion and plans for further work.

II. RELATED WORK

There have been various approaches to speed up deep learning functions. The approaches can be categorized into two groups. In the first group are approaches which try to optimize the size of the neural networks, or, in other words, optimize the software. Approaches in the second group try to optimize the hardware on which neural networks are running. As our approach deals mainly with hardware optimization, we will focus on the related approaches for hardware optimization, and only discuss briefly the advancements in software optimizations.

Because many neural networks, like AlexNet [1], VGG-16 [10], and GoogLeNet [11], have millions of parameters, they are out of the scope of constrained embedded devices, that have small memories and low clock speeds. However, there is much research aimed at developing new neural networks or optimizing existing ones, so that they still work with about the same accuracy, but will not take up as much memory and require too many clock cycles per inference. Significant contributions of this research include the use of pruning [12], quantization [13], and alternative number formats such as 8-bit floating-point numbers [14] or posit [15], [16]. Pruning of the neural network is based on the fact that many connections in a neural network have a very mild impact on the result, meaning that they can simply be omitted. On the other hand, the goal of using alternative number formats or quantization is to minimize the size of each weight. Therefore, if we do not store weights as 32-bit floating-point values, but instead as 16-bit half-precision floating-point values or in an alternative format that uses only 16 or 8 bits (e.g., fixed-point or integer), we reduce the memory requirements by a factor of 2 or 4. In order to make deep learning even more resource-efficient, we can resort to ternary neural networks (TNNs) with neuron weights constrained to \{−1, 0, 1\} instead of full precision values. Furthermore, it is possible to produce binarized neural networks (BNNs) that work with binary values \{−1, 1\} [17]. The authors of [18] showed that neural networks using 8-bit posit numbers have similar accuracy as neural networks using 32-bit floating-point numbers. In [5], it is reported that, by using pruning, quantization, and Huffman coding, it is possible to reduce the storage requirements of neural networks by a factor of 35 to 49.

Much research on neural network hardware focuses on a completely new design of instruction set architectures (ISAs) built specifically for neural networks. The accelerators introduced in [19]–[22], and [23] have the potential to offer the best performance, as they are wholly specialized. The Eyeriss [19] and EIE [20] projects, for example, focus heavily on exploiting the pruning of the neural network, and storing weights in compressed form for minimizing the cost of memory accesses. The authors of [21] also try to optimize memory accesses, but use a different strategy. They conclude that new neural networks are too large to be able to hold all the parameters in a single chip; that is why they use a distributed multi-chip solution, where they try to store the weights as close to the chip doing the computation as possible, in order to minimize the movement of weights. Similarly, in [23], they developed a completely specialized processor that has custom hardware units called Layer Processing Units (LPUs). These LPUs can be thought of as artificial neurons. Before using them, their weights and biases must be programmed, and the
activation functions selected. A certain LPU can compute the output of a particular neuron. This architecture is excellent for minimizing data movement for weights, but limits the size of the neural network significantly. The authors of [22] realized that many neural network accelerator designs lack flexibility. It is why they developed an ISA that is flexible enough to run any neural network efficiently. The proposed ISA has a total of 43 64-bit instructions, which include instructions for data movement and arithmetic computation on vectors and matrices. A similar ISA was developed in [24]. However, because these ISAs are designed specifically for neural networks, they are likely unable to be deployed as a single-chip solution (e.g., a microcontroller is needed to drive the actuators). To lower the cost of the system and save the area on the PCB, we sometimes do not want to use a separate chip to process the neural network.

Other works focus on improving the performance of using CPUs to process neural networks. For example, the authors of [25] show that adding a mixture of vector arithmetic instructions and vector data movement instructions to the instruction set can decrease the dynamic instruction count by 87.5% on standard deep learning functions. A similar instruction set extension was developed by ARM—their new Armv8.1-M [26] ISA for the Cortex-M based devices is extended with vector instructions, instructions for low-overhead loops, and instructions for half-precision floating-point numbers. Unfortunately, as this extension is new, there are currently no results available on performance improvements.

ARM Ltd. published a software library CMSIS-NN [27] that is not tied to the new Armv8.1-M ISA. When running neural networks, CMSIS-NN reduces the cycle count by 78.3%, and it reduces energy consumption by 79.6%. The software library CMSIS-NN achieves these results by using an SIMD unit and by quantizing the neural networks.

A mixed strategy is proposed in [28]. It presents an optimized software library for neural network inference called PULP-NN. This library runs in parallel on ultralow-power tightly coupled clusters of RISC-V processors. PULP-NN uses parallelism, as well as DSP extensions, to achieve high performance at a minimal power budget. By using a neural network realized with PULP-NN on an 8-core cluster, the number of clock cycles is reduced by 96.6% and 94.9% compared with the current state-of-the-art ARM CMSIS-NN library, running on STM32L4 and STM32H7 MCUs, respectively.

Many embedded systems are highly price-sensitive, so the addition of an extra chip for processing neural networks might not be affordable. That is why we optimized our software for a very minimal addition of hardware, which is likely to be part of many embedded processors.

### III. USED HARDWARE AND INSTRUCTION SET EXTENSIONS

For studying the benefit of hardware loops, loop unrolling, and dot product instructions, we used an open-source RISC-V core RISCY [29], also known as CV32E40P. It is a small 32-bit 4-stage in-order RISC-V core, which implements the RV32IMFC instruction set fully. RV32 stands for the 32-bit base RISC-V instruction set, I for integer instructions, M for multiplication and division instructions, F for floating-point instructions, and C for compressed instructions. Additionally, RISCY supports some custom instructions, like hardware loops. Because we used a floating-point network, we extended the core with our floating-point dot product unit. We call this core the modified RISCY core (Fig. 1). It also has an integer dot product unit, but we did not use it. Therefore, it is not shown for the sake of simplicity. RISCY is part of an open-source microcontroller project called PULPino, parts of which we will also use. We call the PULPino microcontroller with the modified RISCY core the modified PULPino. Both the original and the modified RISCY core have 31 general-purpose registers, 32 floating-point registers, and a small 128-bit instruction prefetch cache. Fig. 1 details the RISCY architecture. The non-highlighted boxes show the original RISCY architecture, while the highlighted fDotp box shows our addition. The two orange-bordered boxes are the general-purpose registers (GPR) and the control-status registers (CSR). The violet-bordered boxes represent registers between the pipeline stages. The red-bordered boxes show the control logic, including the Hardware-Loop Controller “hwloop control”, which controls the program counter whenever a hardware loop is encountered (details are explained in Subsection III-A). The gray-bordered boxes interface with the outside world. One of them is the load-store-unit (LSU). The boxes bordered with the light blue color are the processing elements. The ALU/DIV unit contains all the classic arithmetic-logic functions, including a signed integer division. The MULT/MAC unit allows for signed integer multiplication, as well as multiply-accumulate operations. Finally, the fDotp is the unit we added. It is described in Subsection III-B. For more information on the RISCY core and PULPino, we refer the reader to [29], [30], and [31].

![FIGURE 1. The modified RISCY core. Our extension fDotp is highlighted in light blue.](image-url)
involves zero stall clock cycles for jumping from the end to the start of a loop [30], which is why they are more often called zero-overhead loops. As our application contains many loops, we use this feature extensively. The core is also capable of nested hardware loops. However, due to hardware limitation, the nesting is only permitted up to two levels.

Additionally, the instruction fetch unit of the RISC-V core is aware of the hardware loops. It makes sure that the appropriate instructions are stored in the cache. This solution minimizes unnecessary instruction fetches from the main memory.

A hardware loop is defined by a start address, an end address, and a counter. The latter is decremented with each iteration of the loop body [30]. Listing 1 shows an assembly code that calculates the factorial of 5 and stores it in the register x5. Please note that, in RISC-V, x0 is a special register hardwired to the constant 0.

```
addi x5, x0, 1
addi x6, x0, 1
lp.counti x1, 5
lp.start x1, start_HWLP
lp.endi x1, end_HWLP
start_HWLP: mul x5, x5, x6
end_HWLP: addi x6, x6, 1
```

**Listing 1.** Assembly code for calculating the factorial of 5 using hardware loops.

### B. DOT PRODUCT UNIT

To speed up dense arithmetic computation, we added an instruction that calculates a dot product of two vectors with up to four elements, where each element is a single-precision floating-point number (32 bits). The output is a scalar single-precision floating-point number representing the dot product of the two vectors. The dot product unit is shown in Fig. 2. We did not implement any vector load instruction; instead, we used the standard load instruction for floating-point numbers. Consequently, this means that we reused the floating-point register file—saving the area increase of the processor.

The ‘×’ and ‘+’ marks in Fig. 2 represent a floating-point multiplier and a floating-point adder, respectively. The unit performs two instructions, which we added to the instruction set:

- `p.fdotp4.s` - dot product of two 4-element vectors,
- `p.fdotp2.s` - dot product of two 2-element vectors.

When executing the instruction `p.fdotp2.s`, the dot product unit disconnects the terminals of switch S, automatically, and, similarly, it connects them when executing the instruction `p.fdotp4.s`. The RISC-V core runs at a relatively low frequency to reduce energy consumption. Therefore, we can afford that the dot product unit is not pipelined. The result is calculated in a single clock cycle.

**Fig. 2.** A schematic representation of the dot product unit.

**IV. EXPERIMENTAL SETUP**

To test the performance of various deep learning algorithms running on the modified RISC-V core, we developed a testing system. We decided to use a Zynq-7000 System-on-a-Chip (SoC) [32], which combines an ARM Cortex-A9 core and a field-programmable gate array (FPGA) on the same chip. The purpose of the ARM Cortex-A9 core is to program, control, and monitor the modified RISC-V core.

**Fig. 3.** Block diagram of the system.

Fig. 3 shows a block diagram of the system. The diagram is split into two parts—the processing subsystem (PS) and the programmable logic part (PL). On the PL side, there is the emulated modified PULPino, and, on the PS side, the ARM core and the other hard intellectual property cores. In between, various interfaces enable data transfer between both sides, including a universal asynchronous receiver/transmitter interface (UART), a quad serial peripheral interface (QSPI), an advanced extensible interface accelerator coherency port (AXI ACP), and an interrupt line. Note that all blocks in Fig. 3, except the external DDR memory, are in the Zynq-7000 SoC chip.

On the PL side (FPGA) of the Zynq-7000 chip, we emulated not only the RISC-V core, but the entire PULPino microcontroller [33].

The AXI ACP bus enables high-speed data transfers to the microcontroller. In this configuration, we can get the data from the DDR memory, process them, send back the results, and again get new data from the DDR memory. We use the
QSPI bus to do the initial programming of the PULPino memories and UART for some basic debugging.

We designed the system in the Verilog hardware description language using the Vivado Design Suite 2018.2 integrated development environment provided by Xilinx Inc.

V. SOFTWARE

To test the efficiency of the designed instruction set optimization, we developed a simple optical character recognition (OCR) neural network to recognize handwritten decimal digits from the MNIST dataset [34] that contains 60,000 training data and 10,000 test data. The architecture of the neural network is given in Table 1 and Fig. 4. The network was trained in TensorFlow, an open-source software library for numerical computations. Using a mini-batch size of 100, the Cross-Entropy loss function, Adam optimization with a learning rate of 0.01 and 3 epochs of training, recognition accuracy 95% was achieved on the test data. For more information on the neural network, the reader may reference the supplemental material. State-of-the-art neural networks achieve accuracy higher than 99.5% [35]. Compared to them, our neural network performs worse, as it has just one feature map in its only convolutional layer. However, for us, the accuracy of this neural network is not essential, as we only need it to test our hardware.

### TABLE 1. Architecture of the example neural network.

| Layer | Type          | Maps Size | Kernel Stride Size | Activation |
|-------|---------------|-----------|--------------------|------------|
| In    | Input         | 28x28     | -                  | -          |
| C1    | Convolution   | 24x24     | 5x5 1x1            | ReLU       |
| S2    | Max Pooling   | 12x12     | 2x2 2x2            | -          |
| F3    | Fully Connected | 64       | -                  | ReLU       |
| Out   | Fully Connected | 10       | -                  | Softmax    |

![Figure 4. Graphic representation of the example neural network architecture.](image)

The output of the network is a vector of ten floating-point numbers, which represent the probability that the corresponding index of the vector is the digit on the image. In total, the network contains 9,956 parameters, which consume roughly 39 kB of memory space if we use the single-precision floating-point data type. To compute one pass of the network, around 24 thousand multiply-accumulate (MAC) operations must be performed.

A. LOOP UNROLLING

Alongside hardware loops and the dot product unit, we also tested if loop unrolling could benefit the inference performance of our neural network. Loop unrolling is a compiler optimization that minimizes the overhead of loops by reducing or eliminating instructions that control the loop (e.g., branch instructions). This optimization has the side effect of increasing code size.

Algorithm 1 shows a simple for loop that adds up the elements of an array. Algorithm 2 shows the unrolled version of the for loop in Algorithm 1.

#### Algorithm 1 A Simple Standard Loop

```
for i = 0; i < n; i = i + 1 do
    sum = sum + data[i];
end
```

#### Algorithm 2 An Unrolled Loop

```
for i = 0; i < n; i = i + 4 do
    sum = sum + data[i];
    sum = sum + data[i + 1];
    sum = sum + data[i + 2];
    sum = sum + data[i + 3];
end
```

B. DEEP LEARNING LIBRARY

In order to run our network on the modified RISCY, we developed a software library of standard deep learning functions. We first implemented them by using naive algorithms and without using the dot product unit. We named this version of the library the reference version. Following that, we wrote the optimized assembly code that uses the dot product unit and hardware loops. We named this version of the library the optimized assembly version. The naive algorithms were written in C and could also use hardware loops, as the compiler is aware of them. Table 2 lists the functions we implemented in our library. The code can be seen by looking at the supplemental material.

C. FULLY CONNECTED LAYERS

A fully connected layer of a neural network is computed as a matrix-vector multiplication, followed by adding a bias vector and applying a nonlinearity on each element of the resulting vector. In our case, this nonlinearity is the ReLU function. Equation (1) details the ReLU function for scalar input. For a vector input, the function is applied to each element. Equation (2) shows the mathematical operation that
TABLE 2. Functions in our deep learning library.

| Function          | Description                                                                 |
|-------------------|-----------------------------------------------------------------------------|
| dIDense           | Performs matrix-vector multiplication between an arbitrary matrix and a vector and adds the bias. |
| dIDensewReLU      | Performs matrix-vector multiplication between an arbitrary matrix and a vector, adds the bias, and performs the ReLU function over the result vector. |
| dConv2n           | Performs neural network style convolution over an arbitrary input image and a filter. |
| dConv2nwReLU      | Performs neural network style convolution over an arbitrary input image and a filter. It also performs the ReLU function over the output. |
| dMaxPool          | Computes the maximum pooling function.                                       |
| dReLU             | Computes the ReLU function on an arbitrary sized input vector.               |
| dSoftmax          | Computes the softmax function for a given input vector.                     |

is being performed to compute one fully connected layer.

\[
f(x) = \max(0, x) \quad (1)
\]

\[
\vec{r} = f(M \cdot \vec{v} + \vec{b}) \quad (2)
\]

\[M_{m \times k}\] is the matrix of weights of the layer, \([k \times 1]\) is the input vector to the layer, \([m \times 1]\) is the bias term vector, \(f\) is the nonlinear function, and \(\vec{r}_{m \times 1}\) is the output vector, where \(m\) is the number of neurons in the next layer, and \(k\) is the number of neurons in the previous layer or the number of inputs to the neural network.

In the reference version of the deep learning library, we simply used nested for loops to compute the matrix-vector product, and, following that, we applied the ReLU nonlinearity.

The optimized assembly version, however, tries to minimize the number of memory accesses. Because the dot product unit also operates on vectors, let us first clarify the terminology. The matrix and vector on which the matrix-vector multiplication is performed are named the input matrix \(M\) and the input vector \(\vec{v}\), to separate them from the vectors of the dot product unit. Also, the bias vector is denoted by \(\vec{b}\). We load in a chunk of the input vector \(\vec{v}\) and calculate each product for that chunk. It means that we load the vector only once. The size of the chunk is determined by the input size of the dot product; in our case, it is 4. One problem with this approach is that the number of matrix columns must be a multiple of four. This problem can be solved by zero-padding the matrix and vector. Equation (3) illustrates this way of computing the matrix-vector product for a \(4 \times 8\) input matrix \(M\), and an \(8 \times 1\) input vector \(\vec{v}\). Subscripted vectors \(\vec{m}_i^T\), \(i \in \{0, 1, \ldots, 7\}\) and \(\vec{v}_j, j \in \{0, 1\}\), represent the chunks of \(M\) and \(\vec{v}\). Each chunk is a vector with four elements. Each chunk of \(M, \vec{m}_i^T\), is a row vector and the transpose of a column vector \(\vec{m}_i\), because each row of \(M\) is composed of two 4-element row vectors.

\[
M \cdot \vec{v} = \begin{bmatrix} m_0^T & m_1^T & m_2^T & m_3^T \\ m_4^T & m_5^T & m_6^T & m_7^T \end{bmatrix} \cdot \begin{bmatrix} \vec{v}_0 \\ \vec{v}_1 \end{bmatrix} = \begin{bmatrix} m_0^T \cdot \vec{v}_0 + m_1^T \cdot \vec{v}_1 \\ m_2^T \cdot \vec{v}_0 + m_3^T \cdot \vec{v}_1 \\ m_4^T \cdot \vec{v}_0 + m_5^T \cdot \vec{v}_1 \\ m_6^T \cdot \vec{v}_0 + m_7^T \cdot \vec{v}_1 \end{bmatrix} \quad (3)
\]

Fig. 5 shows the placement of the matrix \(M\) and vector \(\vec{v}\) in memory. Each cell in Fig. 5 represents a 32-bit word of memory that contains a single-precision floating-point number. Algorithm 3 shows the pseudocode for executing (3) and applying the ReLU nonlinearity. Functions load_vecA() and load_vecB() take as input the pointer to the memory location from which to load four consecutive scalars. Function load_vecA() is called first—it loads a chunk of the input vector. Next, the inner for loop traverses through a part of the input matrix. Function load_vecB() loads chunks of the input matrix. After loading each chunk, the function dot_prod_vecA_vecB() is called. Algorithm 3 loads data from memory in the following order: It first loads \(\vec{v}_0\), and then vectors \(\vec{m}_0, \vec{m}_1, \vec{m}_2, \vec{m}_3\). Next, it loads \(\vec{v}_1\) and vectors \(\vec{m}_4, \vec{m}_5, \vec{m}_6, \vec{m}_7\). The dot products are computed in the following order: \(m_0^T \cdot \vec{v}_1, m_1^T \cdot \vec{v}_1, m_2^T \cdot \vec{v}_1, m_3^T \cdot \vec{v}_1, m_4^T \cdot \vec{v}_1, m_5^T \cdot \vec{v}_1, m_6^T \cdot \vec{v}_1, m_7^T \cdot \vec{v}_1\). This way, we only have to load vectors \(\vec{v}_0\) and \(\vec{v}_1\) once, and, at the same time, we have a good spatial locality of memory accesses.

D. CONVOLUTIONAL LAYER

Both the input to the convolutional layer and its output are two-dimensional. To compute a pass of a convolutional layer, one must compute what is known in signal processing as a 2D cross-correlation. Following that, a bias is added, and nonlinearity is applied. Equation (4) shows how to compute a single element \(res[x, y]\) of the two-dimensional output.

\[
res[x, y] = f(b + \sum_{i=1}^{out_size} \sum_{j=1}^{out_size} fill[i, j] \cdot \text{img}[x+i, y+j]) \quad (4)
\]
Algorithm 3 Computing a Fully Connected Layer With the ReLU Nonlinearity Applied (dlDense nwReLU)

\[
\begin{array}{l}
\text{input : pointer to matrix } M \text{ of size } m \times k \\
\text{input : pointer to vector } v \text{ of size } k \\
\text{input : pointer to bias } b \text{ of size } m \\
\text{output: pointer to result vector } r \text{ of size } m \\

\text{for } i = 0 \text{ to } (k/4) - 1 \text{ do} \\
\quad \text{load_vecA}(v + i \times 4); \\
\quad \text{for } j = 0 \text{ to } m - 1 \text{ do} \\
\quad\quad \text{load_vecB}(M + i \times m * 4 + j \times 4); \\
\quad\quad r[j] = r[j] + \text{dot_prod_vecA_vecB}(); \\
\quad \end{array}
\]

end

\[
\begin{array}{l}
\text{for } i = 0 \text{ to } m - 1 \text{ do} \\
\quad r[i] = r[i] + b[i]; \\
\quad \text{if } r[i] < 0 \text{ then} \\
\quad\quad r[i] = 0; \\
\quad \end{array}
\]

end

\[
\begin{array}{l}
\text{end}
\end{array}
\]

\[
\text{fil} \text{ is a two-dimensional filter of size } \text{fil}_\text{size} \times \text{fil}_\text{size}, \text{img} \text{ is a two-dimensional input array of size } \text{img}_\text{size} \times \text{img}_\text{size}, \text{b} \text{ is a scalar bias term, } \text{res} \text{ is the output of size } \text{out}_\text{size} \times \text{out}_\text{size}, \text{and } f \text{ is the nonlinear function. Note that more complicated convolutional neural networks typically have three-dimensional filters. We show a two-dimensional case for presentation, but to handle three dimensions, we simply repeat the procedure for the two-dimensional case.}
\]

The reference version of the function dlConv2nwReLU simply uses nested for loops to compute the convolution in the spatial domain. It then calls the ReLU function on the output matrix. Our optimized version tries to optimize the number of memory accesses. We do that by always keeping the filter in the register file. We first save the contents of the register file to the stack. Such an approach enables us to use the entire register file without breaking the calling convention used by the compiler. Next, we load the entire \(5 \times 5\) filter and the bias term into the floating-point register file. RISC-V has 32 floating-point registers, so we have enough room in the register file to store the \(5 \times 5\) filter, a chunk of the image, and still have two registers left. Note that we again use the word chunk to refer to four floating-point numbers. Fig. 6(a) and Fig. 6(b) show how we store the \(5 \times 5\) filter and the bias term in memory, and load them into the floating-point register file. Registers f28-f31 are used to store chunks of the image and registers f9 and f10 to store the result.

Having the filter and bias term in the register file, we load in one chunk of the image at a time and compute the dot product with the appropriate part of the filter already stored in the register file. After traversing the entire image, we restore the previously saved register state. We make use of hardware loops to implement looping behavior.

Computing the convolutional layer is shown in detail in Algorithm 4. The functions load_vec f* load a total of 4 consecutive floating-point numbers from the memory location given in the argument to the registers f* to f* + 3. The function load_f* loads a single floating-point number to register f*. The functions dot_product_f* f* f$ compute the dot product between two chunks in the register file. The first one starts at f* and ends at f* + 3, and the second one starts at f$ and ends at f$ + 3.

Fig. 7 shows Algorithm 4 in action at the moment after the first iteration of the inner for loop. The leftmost matrix represents the input image, the middle matrix is the filter, and the rightmost matrix is the result matrix. Note that, in Fig. 7, the input image and filter contain only ones and zeros, so that anyone can calculate the dot product result quickly (8 in our case) in the upper-left corner of the result matrix by mental arithmetic. In fact, there are floating-point numbers in each cell.

VI. RESULTS

We provide a thorough set of results to give the reader a full picture of the costs and benefits of our proposed instruction set extension. All results of percentage decreases and increases according to the baseline values are rounded to whole numbers.

A. SYNTHESIS RESULTS

The synthesis was run using the Synopsys Design Compiler O-2018.06-SP5 and the 90 nm generic core cell library from the United Microelectronics Corporation.
### Algorithm 4 Computing the Convolutional Layer With the ReLU Nonlinearity (dlConv2nwReLU)

**input**: pointer to image `img` of size `img_size` × `img_size`
**input**: pointer to a filter `fil` of size `5` × `5`
**input**: bias `b` of size `1`
**input**: stride `stride`
**output**: result `res` of size `out_size` × `out_size`

```c
saveRegisterState();
loadFilterAndBiasToRegister();
out_size = ((img_size − fil_size)/stride) + 1;
// x and y get incremented by stride after each // iteration
for x = 0 to out_size do
    for y = 0 to out_size do
        img_slice ← img[x][y];
        load_vec_f28_31(img_slice[0][0]);
        res[x][y] = dot_prod_f0_f28();
        load_vec_f28_31(img_slice[1][0]);
        res[x][y] = res[x][y] + dot_prod_f4_f28();
        load_vec_f28_31(img_slice[2][0]);
        res[x][y] = res[x][y] + dot_prod_f12_f28();
        load_vec_f28_31(img_slice[3][0]);
        res[x][y] = res[x][y] + dot_prod_f16_f28();
        load_vec_f28_31(img_slice[4][0]);
        res[x][y] = res[x][y] + dot_prod_f20_f28();
        load_f28(img_slice[0][4]);
        load_f29(img_slice[1][4]);
        load_f30(img_slice[2][4]);
        load_f31(img_slice[3][4]);
        res[x][y] = res[x][y] + dot_prod_f24_f28();
        load_f28(img_slice[4][4]);
        res[x][y] = res[x][y] + dot_prod_f8_f28();
        res[x][y] = res[x][y] + f11;
        if res[x][y] < 0 then
            res[x][y] = 0;
        end
    end
restoreRegisterState();
```

### Table 3. Synthesis results.

| Area¹ [kGE] | Area [µm²] | Leakage power [µW] | Dynamic power [mW] |
|-------------|------------|--------------------|--------------------|
| Original RISCY | 77,274.50 | 242,332.83 | 307.13 | 147.78 |
| Modified RISCY | 133,159.50 | 417,588.19 | 640.37 | 148.47 |

¹ kGE signifies kilo-gate equivalent

Table 3 shows the results of the synthesis. We see that the area of the modified RISCY is 72% larger than the original RISCY. The area increase is only due to the addition of the dot product unit, not the hardware loops. The price in the area of adding hardware loops is minor, about 3 kGE [29]. Since the RISCY core already has a floating-point unit, we could reduce the area increase by reusing one floating-point adder and one floating-point multiplier in the dot product unit.

Dynamic power consumption was reported by the Design Compiler using random equal probability inputs, so these results are only approximate. The leakage power consumption has more than doubled, but the dynamic power consumption has increased only slightly. It is important to note that the dynamic power consumption is three orders of magnitude higher, so the total power is still about the same. However, the rise in leakage power might be concerning for some low-power embedded systems, that stay most of the time in standby mode. This concern can be addressed by turning off the dot product unit while in standby mode.

### B. METHODOLOGY

To gather data about the performance, we used the performance counters embedded in the RISCY core. We compared and analyzed our implementation in the following metrics:

- **Cycles**—number of clock cycles the core was active,
- **Instructions**—number of instructions executed,
- **Loads**—number of data memory loads,
- **Load Stalls**—number of load data hazards,
- **Stores**—number of data memory stores,
- **Jumps**—number of unconditional jump instructions executed,
- **Branch**—number of branches (taken and not taken),
- **Taken**—number of taken branches.

We compared five different implementations, listed in Table 4. All of them computed the same result. The implementations Fp, FpHw, and FpHwU use the reference library, and implementations FpDotHw and FpDotHwU the optimized assembly library. We chose the Fp implementation as the baseline. Our goal was to find out how much the hardware loops, loop unrolling, and the dot product unit aided in speeding up the computation. The dot product unit is not used in all functions of our library, but only in dlDense, dlDenseFwReLU, dlConv2n, and dlConv2nwReLU. However, these functions represent most of the computing effort.

For compiling our neural network, we used the modified GNU toolchain 5.2.0 (riscv32-unknown-elf-). The modified toolchain (riscy-gnu-toolchain) is provided as part of the PULP platform. It supports custom extensions such as hardware loops, and applies them automatically when compiling the neural network with optimizations enabled. Hardware loops can be disabled explicitly using the compiler flag `-mnohwloop`. The neural networks were compiled with the compiler flags that are listed in Table 4. Even though the RISCY core supports compressed instructions, we did not make use of them.
TABLE 4. Different implementations of the program.

| Implementation | GCC compiler flags | Description |
|----------------|--------------------|-------------|
| Fp             | -O3 -m32 -g -mnohloop | Reference library version with all optimizations except hardware loops. |
| FpHw           | -O3 -m32 -g         | Reference library version with all optimizations turned on. |
| FpHwU          | -O3 -m32 -g -funroll-all-loops | Reference library version with all optimizations and loop unrolling. |
| FpDotHw        | -O3 -m32 -g         | An optimized assembly library version that uses the dot product unit and all optimizations turned on for functions that are not written in inline assembly. |
| FpDotHwU       | -O3 -m32 -g -funroll-all-loops | An optimized assembly library version that uses the dot product unit and all optimizations and loop unrolling turned on for functions that are not written in inline assembly. |

C. CODE SIZE COMPARISON

We compared the size of the code for the functions listed in Table 2. The results are shown in Table 5. The first three implementations (Fp, FpHw, and FpHwU) use the reference version of the library, while implementations FpDotHw and FpDotHwU use the optimized assembly version of the library. The assembly code for a particular (hardware) implementation is called a function implementation. Note that not all function implementations were written in inline assembly, but only the ones whose code sizes are highlighted blue in Table 5—these function implementations are not affected by compiler optimizations, and, because of that, they are identical at the assembly level. For example, the implementations FpDotHw and FpDotHwU use exactly the same code for the first four functions. On the other hand, the implementations FpHwU and FpDotHw have the same code size for the function dlDensen, but the codes are not identical. It is just a coincidence. Each of the last three functions (dlMaxPool, dlReLU, and dlSoftmax) has an identical function implementation and code size in FpHw and FpDotHw, as well as in FpHwU and FpDotHwU, because the same compiler optimizations apply for implementations with or without a dot product unit. Different functions may be of the same size, as are the sizes of dlConv2n and dlConv2nwReLU for the implementation Fp. Nevertheless, these function implementations are not identical. The reason for the same size is the fact that the ReLU functionality in dlConv2nwReLU is implemented by calling the dlReLU function, which does not affect the size of the function dlConv2nReLU.

TABLE 5. Code size of functions in our deep learning library. All results are in bytes.

| Functions         | Fp   | FpHw  | FpHwU | FpDotHw | FpDotHwU |
|-------------------|------|-------|-------|---------|----------|
| dlDensen          | 148  | 192   | 352   | 352     | 352      |
| dlDensenReLU      | 168  | 208   | 696   | 360     | 360      |
| dlConv2n          | 196  | 200   | 576   | 912     | 912      |
| dlConv2nwReLU     | 196  | 200   | 576   | 200     | 200      |
| dlMaxPool         | 172  | 180   | 568   | 180     | 568      |
| dlReLU            | 56   | 68    | 472   | 68      | 472      |
| dlSoftmax         | 248  | 256   | 1188  | 256     | 1188     |

In the Fp implementation, where neither hardware loops nor loop unrolling are used, the code size is the smallest. As is predictable, the code with loop unrolling (including our optimized inline assembly code) is the largest. Nevertheless, the code size is still quite small, and in the realm of what most microcontrollers can handle.

D. FULLY CONNECTED LAYERS

Let us first look at the results for fully connected layers. These layers compute the matrix-vector product, add the bias and apply the ReLU function. Matrix-vector multiplication takes most of the time. The reader should keep in mind that computing a matrix-vector product is also very memory intensive.

What we are measuring is computing the F3 layer of our example neural network shown in Table 1. The matrix-vector product consists of a matrix with a dimension 64 × 144 and a column vector of 144 elements.

We ran the same code twenty times with different inputs, and computed the average of the runs. The averages are displayed in Fig. 8(a) and Fig. 8(b).

Fig. 8(a) compares the number of clock cycles needed to compute the fully connected layer. Hardware loops alone contribute to a 29% reduction in clock cycles compared to the baseline. A decrease of 72% was achieved with the dot product unit included. The result makes sense, since we replaced seven instructions (four multiplications and three additions) with just one. It would be even better if we could feed the data to the RISCY core faster. Let us conclude as follows. With our hardware, it takes only one clock cycle to calculate a single dot product of two vectors of size four, but at least eight clock cycles are needed to fetch the data for this dot product unit. Because the dot product is calculated over two vectors of size four and each access takes one clock cycle, our dot product unit is utilized only 11% of the time. This result is calculated in (5).

\[
\text{Cycles dot product unit is used} = \frac{1}{8 + 1} = 11% \quad (5)
\]

The actual utilization is slightly higher, because we reuse one vector (a chunk) several times, as seen in Algorithm 3. This fact means that the best possible utilization of our dot product unit is \(1/(4 + 1) = 20\%\).
Using loop unrolling does not provide any benefit in the implementation, neither with nor without the dot product unit. It slows down FpHwU, and FpDotHwU has about the same amount of clock cycles as FpDotHw. The implementation using hardware loops and loop unrolling (FpHwU) achieved a 21% reduction in clock cycle count compared to the baseline, but a 29% reduction was achieved using only hardware loops (FpHw). The reason is that loop unrolling makes the caching characteristics of the code much worse (the cache is no longer big enough). This effect can be seen by looking at the number of load stalls for the FpHwU implementation in Fig. 8(c).

The dynamic instruction count comparison can be seen in Fig. 8(b). Hardware loops contribute a 13% reduction in dynamic instruction count compared to the baseline. The optimized inline assembly code for the dot product unit (FpDotHw) contributed a 63% reduction of the baseline dynamic instruction count. It is a predictable consequence of having one single instruction that calculates a dot product of two 4-element vectors instead of seven scalar instructions. Loop unrolling reduces the dynamic instruction count. Hardware loops, combined with loop unrolling (FpHwU), contributed a 22% reduction in the dynamic instruction count compared to the baseline. Unrolling the loop reduces the number of loop iterations, but it does not make sense for hardware loops because there is no overhead. However, not all loops can be made into hardware loops due to the limitations of the RISCY core.

The results of auxiliary performance counters are shown in Fig. 8(c). Our inline assembly code for the dot product unit (FpDotHw) reduces the number of branches substantially. Our algorithm, combined with the instruction extensions, reduced the number of loads and stores—as can be seen in Fig. 8(c). The number of loads and stores in the FpDotHw implementation was reduced by 36% and 74%, respectively compared to the baseline.

E. CONVOLUTIONAL LAYER

In this Subsection we look at the cost of computing the C1 layer from Table 1. It is a convolution of a $28 \times 28$ input picture with a $5 \times 5$ filter. We also include the cost of computing the ReLU function on the resulting output. We do this because we integrated the ReLU function into the convolution algorithm in order to optimize the code (dlConv2nwReLU). This layer differs from the fully connected layer, as it is not so constrained by memory bandwidth. It means that we can use our dot product unit more effectively. As in the case of the fully connected layer, we were not able to fetch data fast enough to utilize the unit entirely. It can be seen in Fig. 9(a) that using our dot product unit (FpDotHw) contributed to a 78% reduction of the clock cycle count compared to the baseline implementation (Fp). Adding just hardware loops to the instruction set (FpHw) contributed only a 23% reduction. As in the case of the fully connected layers, loop unrolling was not effective.

The dynamic instruction count was again substantially lower when using the dot product unit. This fact is a consequence of dot product instruction. By using it, seven instructions are replaced with only one. In Fig. 9(b) we see that the dynamic instruction count of the FpDotHw implementation is reduced by 72% compared to the baseline.
dynamic instruction count. Hardware loops alone (FpHw) have a modest impact. They contribute only a 9% reduction in the dynamic instruction count. Compared to the baseline implementation (Fp), loop unrolling increases the dynamic instruction count by 7% for computing a convolutional layer when not using a dot product unit (FpHwU). The dynamic instruction count is increased because of the increase in the number of branches, and it is a consequence of unrolling loops with branches inside them (e.g., function dlReLU).

The results of auxiliary performance counters are shown in Fig. 9(c). The FpDotHw and FpDotHwU implementations reduced the number of loads by 50% compared to the baseline. The number of stores was reduced by 95%. Both results mentioned above are a consequence of our strategy to keep the entire filter and bias term in the register file. Hardware loops again decreased the number of branch instructions. The number of branches in the FpHw implementation was reduced by 91% compared to the baseline. Since we used many small loops, hardware loops are a good idea. The FpDotHw implementation reduced the number of branches even further—by 97% compared to the baseline. Namely, the only branches in the FpDotHw implementation come from the ReLU function, while the FpHw function has additional branches for non-hardware loops. As is the case of fully connected layers, the number of branches increases significantly if loop unrolling is used.

F. ENTIRE NEURAL NETWORK

Finally, let us look at the results of running the entire example neural network. Fig. 10(a) shows that the FpDotHw implementation reduces the clock cycle count by 73% compared to the baseline. Hardware loops alone (FpHw) reduce the clock cycle count only by 24%. If we ran our microcontroller at only 10 MHz, we could run a single inference of the neural network in 7.5 ms using the FpDotHw implementation.

From Fig. 10(b) we can see that the dynamic instruction count for the FpDotHw implementation is reduced by 66% of the baseline version. The FpHw version also reduces the dynamic instruction count, but only by 10%.

Fig. 10(c) shows the average results of the auxiliary performance counters for running a single-pass of the entire neural network. In general, we can say that, by using the dot product unit and the optimized inline assembly code (the FpDotHw implementation), we reduced the number of stores by 86%, and the number of loads by 43% compared to the baseline (Fp). Since neural networks are very data-intensive, this result is auspicious.

G. ENERGY CONSUMPTION

We derived the energy consumption results from theASIC synthesis power results and the number of the executed clock cycles for the various implementations. Fig. 11 shows the energy results of running our entire neural network. In general, we can say that, by using the dot product unit and the optimized inline assembly code (the FpDotHw implementation), we reduced the number of stores by 86%, and the number of loads by 43% compared to the baseline (Fp). Since neural networks are very data-intensive, this result is auspicious.
by using the power consumption results of the original RI5CY (see Table 3). For the implementations FpDotHw and FpDotHwU, energy consumptions were calculated by using the power consumption results of the modified RI5CY. Note that the results do not include the energy consumption of data transactions to and from the main memory.

Our reduction in cycle count is comparable with the 78.3% reduction achieved by CMSIS-NN [27]. Similarly, the combination of hardware loops and dot product instructions reduced the dynamic instruction count by 66%. Although our reduction in dynamic instruction count is less than the 87.5% reduction gained in [25], we achieved this reduction with a considerably smaller ISA extension. As embedded systems are highly price-sensitive, this is an important consideration. Unfortunately, in [25], the ISA extension hardware cost is not discussed. A point to emphasize is that our ISA improvements for embedded systems should be considered together with other research on compressing neural networks. Getting the sizes of neural networks down is an essential step in expanding the possibilities for neural networks in embedded systems. For example, in [5], it is shown that it is possible to quantize neural networks to achieve a size reduction of more than 90%. Another interesting topic for further research is Posit—an alternative floating-point number format, that may offer additional advantages, as it has an increased dynamic range at the same word size [15], [36]. Because of the improved dynamic range, weights could be stored in lower precision, thus, again, decreasing the memory requirements. Combining the reduced size requirements with low-cost ISA improvements could make neural networks more ubiquitous in the price-sensitive embedded systems market.

VII. CONCLUSION
The main aim of our research was to evaluate the effectiveness of hardware loop instructions and the dot product instructions for speeding up neural network computation. We showed that hardware loops alone contributed a 24% cycle count decrease, while the combination of hardware loops and dot product instructions reduced the clock cycle count by 73%.

ACKNOWLEDGMENT
The authors would like to acknowledge the contributions of Dr. Thomas Grass, whose assistance in actual implementation and debugging helped to complete the research successfully.
REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” Commun. ACM, vol. 60, no. 6, pp. 84–90, Jun. 2017.

[2] S. Kodali, P. Hansen, N. Mulholland, P. Whatmough, D. Brooks, and G.-Y. Wei, “Applications of deep neural networks for ultra low power IoT,” in Proc. IEEE Int. Conf. Comput. Design (ICCD), Boston, MA, USA, Nov. 2017, pp. 589–592.

[3] M. Hügie, S. Heller, M. Watter, M. Blum, F. Manzouri, M. Dömpfl, A. Schulze-Bonhage, P. Woias, and J. Boedecker, “Early seizure detection with an energy-efficient convolutional neural network on an implantable microcontroller,” in Proc. Int. Joint Conf. Neural Netw. (IJCNN), Rio de Janeiro, Brazil, Jul. 2018, pp. 1–7. [Online]. Available: https://arxiv.org/abs/1806.04549

[4] Z. Ghodsi, T. Gu, and S. Garg, “SafetyNet: Verifiable execution of deep neural networks on an untrusted cloud,” in Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS), Long Beach, CA, USA, Dec. 2017, pp. 4675–4684.

[5] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in Proc. 4th Int. Conf. Learn. Reproc. (ICLRR), San Juan, Puerto Rico, May 2016, pp. 1–14. [Online]. Available: http://arxiv.org/abs/1510.00149

[6] E. Liberis and N. D. Lane, “Neural networks on microcontrollers: Saving memory at inference via operator reordering,” 2019, arXiv:1910.05110. [Online]. Available: http://arxiv.org/abs/1910.05110

[7] U. Thakkher, J. Beu, D. Gope, C. Zhou, I. Fedorov, G. Dusika, and M. Mattina, “Compressing RNNs for IoT devices by 15-38x using Kronecker products,” 2019, arXiv:1906.02876. [Online]. Available: http://arxiv.org/abs/1906.02876

[8] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough, “SpASe: Sparse architecture search for CNNs on resource-constrained microcontrollers,” 2019, arXiv:1905.12107. [Online]. Available: http://arxiv.org/abs/1905.12107

[9] K. L. Lueth. (Aug. 8, 2018). State of the IoT 2018: Number of IoT Devices Now at 7B—Market Accelerating. IoT Analytics. Accessed: Jun. 8, 2020. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

[10] J. Gustafson and I. Yonemoto, “Beating floating point at its own game: VOLUME 8, 2020

[11] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, Y. Lecun, J. Denker, and S. Solla, “Optimal brain damage,” in Proc. Adv. Neural Inf. Process. Syst. (NIPS), 2015, vol. 26, no. 7, pp. 1408–1412, Jun. 8, 2020. [Online]. Available: https://www.pulp-platform.org/index.html

[12] M. S. Louis, Z. Azad, L. Delshadehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, “Towards deep learning using TensorFlow Lite on RISC-V,” in Proc. 3rd Workshop Comput. Archit. Res. RISC-V (CARRV), Phoenix, AZ, USA, Jun. 2019, pp. 1–6.

[13] J. Yu, “Introduction to Armv8.1-M architecture,” ARM, Cambridge, U.K., Feb. 2019. [Online]. Available: https://pages.arm.com/en-si/312-SAX-488/images/Introduction_to_Armv8.1-M_architecture.pdf

[14] L. Nai, S. Luda, and V. Chandra, “CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs,” 2018, arXiv:1801.06601. [Online]. Available: http://arxiv.org/abs/1801.06601

[15] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, “PULP-NN: Accelerating quantized neural networks on parallel ultra-low-power RISC-V processors,” Phil. Trans. Roy. Soc. A, Math., Phys. Eng. Sci., vol. 378, no. 2164, Feb. 2020, Art. no. 20190155.

[16] M. Gauthsi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, “Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices,” IEEE Trans. Very Large Scale Inte. (VLSI) Syst., vol. 25, no. 10, pp. 2700–2713, Oct. 2017.

[17] A. Traber, M. G. Pasquale, and D. Schiavone, “RISCY: User manual,” Micrel Lab Multitherman Lab, Univ. Bologna, Italy, Integr. Syst. Lab, ETH Zürich, Zürich, Switzerland, Apr. 2019. Accessed: Jun. 8, 2020. [Online]. Available: https://www.pulp-platform.org/docs/riscy_user_manual.pdf

[18] A. Traber and M. Gauthsi, “PULPino: Datasheet,” ETH Zürich, Zürich, Switzerland, Jun. 2017. Accessed: Jun. 8, 2020. [Online]. Available: https://pulp-platform.org/docs/pulpino_datasheet.pdf

[19] Xilinx. Zynq7000 SoC, Accessed: Jun. 8, 2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq7000.html

[20] T. P. team. (2019). PULP Platform. Accessed: Jun. 8, 2020. [Online]. Available: https://pulp-platform.org/index.html

[21] Y. LeCun. (2019). The MNIST Database of Handwritten Digits. Accessed: Jun. 8, 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[22] D. Ciresan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., Providence, RI, USA, Jun. 2012, pp. 3642–3649.

[23] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers, “Parameterized posicast arithmetic hardware generator,” in Proc. IEEE 35th Int. Conf. Comput. Des. (ICCD), Orlando, FL, USA, Oct. 2018, pp. 334–341.

JURE VREČA received the B.Sc. degree in electrical engineering from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 2017, where he is currently pursuing the M.Sc. degree in electrical engineering. During his postgraduate study, he did a six-month visit to the Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany. His research interests include computer architecture, embedded system design, and digital integrated circuits.
KARL J. X. STURM (Student Member, IEEE) is currently pursuing the bachelor’s degree with the Computer Engineering Program, RWTH Aachen University. His research interests include hardware-software co-design and high-performance computing.

ERNEST GUNGL (Member, IEEE) received the M.Sc. degree in electrical engineering from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 2007. He is currently pursuing the Ph.D. degree in electrical engineering. In 2014, he joined the faculty as a Research Assistant. His main research interest includes electronics and machine learning.

FARHAD MERCHANT received the Ph.D. degree from the Indian Institute of Science, Bengaluru, India, in 2016. His Ph.D. thesis title was Algorithm-Architecture Co-Design for Dense Linear Algebra Computations. He worked as a Postdoctoral Research Fellow at Nanyang Technological University (NTU), Singapore, from March 2016 to December 2016. In December 2016, he moved to Corporate Research in Robert Bosch, Bengaluru, as a Researcher, where he worked on numerical methods for ordinary differential equations. He joined the Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, in December 2017, as a Postdoctoral Research Fellow of Chair for Software for Systems on Silicon. He was a recipient of the HiPEAC Technology Transfer Award, in 2019, and the DAAD Fellowship during his Ph.D.

PAOLO BIENTINESI received the Laure degree in computer science from the University of Pisa, Italy, in 1998, and the Ph.D. degree from The University of Texas at Austin, USA, in 2006. He was the Deputy Director of the Aachen Institute for Advanced Study in Computational Engineering Science. He is currently a Professor of high-performance computing with Umeå University, Sweden. Before moving to Umeå, he spent two years as a Postdoctoral Researcher at Duke University, USA, and 11 years as a Professor at RWTH Aachen University, Germany, where he established and led an Intel Parallel Computing Center and the Research Group High-Performance and Automatic Computing. His research interests include multiple domains, including numerical linear algebra, performance modeling, automatic algorithm and code generation, and computer music. He is keen on interdisciplinary research and a strong supporter of reproducibility. He has published more than 80 peer-reviewed articles, most of his contributions are complemented by open-source code and libraries. He won the 2009 Karl Arnold Prize from the North Rhine-Westphalian Academy of Sciences and Humanities for outstanding research work.

RAINER LEUPERS received the M.Sc. (Dipl.-Inform.) and Ph.D. (Dr. rer. nat.) degrees (Hon.) in computer science from TU Dortmund, in 1992 and 1997, respectively. From 1997 to 2001, he was the Chief Engineer with the Embedded Systems Chair, TU Dortmund. In 2002, he joined RWTH Aachen University, as a Professor for Software for Systems on Silicon. His research interests include embedded software development tools, multicore processor architectures, hardware security, and system-level electronic design automation. He is also engaged as an Entrepreneur and in turning research results into innovations. He holds several patents and has been a co-founder of LISAtek (now Synopsys), Silexica GmbH, and Secure Elements. As the coordinator of the TETRACOM and TETRAMAX projects, he contributes to EU-wide academia-to-industry technology transfer. He received various scientific awards, including Best Paper Awards at DAC and twice at DATE and several industrial awards. He has served on committees of the leading international EDA conferences.

ZMAGO BREZOČNIK (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 1986 and 1992, respectively. He was the Vice-Dean of Education and the Head and Deputy Head of the Institute of Electronics and Telecommunications. In 1993, he founded the IEEE University of Maribor Student Branch, and served as its Counselor, until 2002. He is currently a Full Professor and the Head of the Laboratory for Microcomputer Systems. His main research interests include formal methods and tools for software and protocol verification, especially model checking, binary decision diagrams, and digital system design. He is the leading author of SpinRCP—a freely available integrated development environment for the Spin model checker. He was a member of the organizing and program committees of several international conferences and workshops.