IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received 7 December 2022, accepted 26 December 2022, date of publication 5 January 2023, date of current version 12 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3234622

RESEARCH ARTICLE

Most Resource Efficient Matrix Vector Multiplication on FPGAs

ALEXANDER LEHNERT[1], PHILIPP HOLZINGER[2], SIMON PFENNING[2], RALF MÜLLER[3], (Fellow, IEEE), AND MARC REICHENBACH[1], (Member, IEEE)

[1]Chair of Computer Engineering, Brandenburg University of Technology Cottbus-Senftenberg, 03046 Cottbus, Germany
[2]Chair of Computer Architecture, Friedrich-Alexander University Erlangen-Nürnberg, 91058 Erlangen, Germany
[3]Institute for Digital Communications, Friedrich-Alexander University Erlangen-Nürnberg, 91058 Erlangen, Germany

Corresponding author: Alexander Lehnert (alexander.lehnert@b-tu.de)

This work was supported in part by the German Research Foundation Deutsche Forschungsgesellschaft (DFG) through the Project Berechnungscodierung under Grant RE 4182/4-1 and Grant MU 3735/8-1.

ABSTRACT Fast and resource-efficient inference in artificial neural networks (ANNs) is of utmost importance and drives many new developments in the area of new hardware architectures, e.g., by means of systolic arrays or algorithmic optimization such as pruning. In this paper, we present a novel method for lowering the computation effort for ANN inference utilizing ideas from information theory. Weight matrices are sliced into submatrices of logarithmic aspect ratios. These slices are then factorized. This reduces the number of required computations without compromising on fully parallel processing. We create a new hardware architecture for this dedicated purpose. We also provide a tool to map these sliced and factorized matrices efficiently to reconfigurable hardware. By comparing to the state of the art FPGA implementations, we can prove our claim by lowering hardware resources measured in look-up-tables (LUTs) by a factor of three to six. Our method does not rely on any particular property of the weight matrices of the ANN. It works for the general task of multiplying an input vector with a constant matrix and is also suitable for digital signal processing beyond ANNs.

INDEX TERMS Constant matrix multiplication, neural networks, computer architecture, reconfigurable architectures, computational efficiency.

I. INTRODUCTION

Artificial Neural Networks (ANNs) are widely used today in different application fields such as image processing [1], [2], [3], [4], speech recognition [5], [6], [7] or predictive maintenance [8], [9]. Compared to classical signal processing algorithms, they can achieve a very high classification quality without manual design of handcrafted algorithms. While these outstanding features will enable to solve even more and more complex problems, computational effort of such ANNs could become very large and energy intensive. This is especially true for inference in ANNs which mainly relies on constant matrix vector multiplications (CMVMs) and is the focus of this paper. Throughout the paper the terms CMVM and constant matrix multiplication (CMM) are used interchangeably. Next to ANNs also many digital signal processing (DSP) algorithms rely heavily on CMMs [10], [11]. In the past, there were efforts to improve the computational complexity of these operations [12], and also approximate methods were explored [13].

With the goal of area, as well as power efficient architectures implementing CMMs of ANNs or DSP algorithms, several approaches in the past were researched. Then can roughly be divided into the three following domains.

1) Algorithm optimization such as quantization (use numbers with a limited bit width) and pruning (e.g. the complete removal of neurons)
2) Specialized dataflow architectures such as systolic arrays or coarse-grained reconfigurable arrays
3) Advances in technology such as crossbar arrays or memristive memory cells

These approaches can be combined in a smart way, e.g. with 1) the ANN is modified in a pre-defined way, which an architecture 2) can utilize as a priori knowledge to build very fast accelerator architectures. This is also true for DSP

The associate editor coordinating the review of this manuscript and approving it for publication was Alireza Sadeghian.

algorithms which are mainly using fixed matrices and thus provide even more application-independent a priori knowledge. We present 1) a decomposition algorithm to restructure matrices based on a priori knowledge to then 2) provide an architecture that makes use of the restructured information to lower hardware cost while offering a high throughput. With the goal of high throughput in mind, we refer to an efficient design as one that is 1) resource-aware, i.e. minimizes hardware cost, while maximizing 2) throughput and 3) energy-awareness. Architectures are designed fully rolled-out to preserve high throughput and are compared as such. Competing designs that exist for ANNs are, e.g., FINN [14], a design framework for quantized neural networks, and for general CMVM, e.g., approaches based on Canonically Signed Digit (CSD) representation [13].

To optimize computation effort in ANNs, a close look to their internal structure is necessary: The architecture of an ANN consists of several layers. For the inference of an ANN, the equation

$$\boldsymbol{a} = \phi(\mathbf{W}\boldsymbol{v} + \boldsymbol{b}) \tag{1}$$

has to be solved for **each** layer. Here and in the following, $\mathbf{W}$ denotes the weight matrix, $\boldsymbol{v}$ the input vector, $\boldsymbol{a}$ the output vector, $\boldsymbol{b}$ the bias vector, and ϕ the so-called activation function. While in current ANNs, the scalar functions ϕ involve low computation effort (e.g. rectified linear unit (ReLU)), as they operate element-wise, the matrix-vector multiplication $\mathbf{W}\boldsymbol{v}$ remains computationally intensive. Therefore, we will present in this paper a novel approach to optimize exactly this calculation which also can be directly applied to DSP algorithms based on CMMs. For this purpose, we solve this problem also on the above-mentioned two levels, i.e. 1) the algorithmic level and 2) provide a dedicated hardware architecture.

The basic idea we propose is to vertically slice the unrestricted matrix into S submatrices

$$\mathbf{W} = [\mathbf{W}_1|\mathbf{W}_2|\ldots|\mathbf{W}_S] \tag{2}$$

which are subsequently factorized into P matrix factors as proposed in [15]

$$\mathbf{W}_s \approx \mathbf{F}_{s,P} \cdots \mathbf{F}_{s,1}\mathbf{F}_{s,0}. \tag{3}$$

In the application of ANNs, the matrices that are decomposed are the weight matrices. As we will discuss in Section III in full details, this decomposition (so-called computation coding) will bring the following advantages:

- The matrices $\mathbf{F}_{1,1}$ to $\mathbf{F}_{S,1}$ do not require computations at all.
- The matrices $\mathbf{F}_{1,1}$ to $\mathbf{F}_{S,P}$ will be sparse with a well defined structure.
- The matrices $\mathbf{F}_{1,1}$ to $\mathbf{F}_{S,P}$ will only contain numbers related to a power of two.

The matrices $\mathbf{F}_{1,1}$ to $\mathbf{F}_{S,P}$ are sparse and contain only signed powers of two such that the multiplication of any of them with a vector requires only a fixed number of additions per row. Furthermore, all those additions can be executed, in parallel. From now on we will refer to such well-behaved matrices as *CC-matrices*, referring to the computation coding (CC) decomposition algorithm which they originate from.

As firstly described in [16] and [15], we can transform a matrix into a set of CC-matrices. All these new matrices are well-behaved meaning each row of them features a fixed number of non-zero entries per row which are signed powers of two. When implementing a matrix-vector product architecture the well-behaved property of the underlying CC matrices leads to a lower computational effort because no multiplications are needed anymore as they can be replaced by shifts. Moreover, the prior knowledge of the structure of $\mathbf{F}_{1,1}$ to $\mathbf{F}_{S,P}$ will enable the creation of dedicated hardware circuits, which perfectly utilize this approach. Nevertheless, as shown in (3), this transformation will introduce a small error. Fortunately, similar to any fixed point arithmetic the error can be determined and lowered arbitrarily well which is important for the overall accuracy of the ANN inference.

While in [16] and [15], the basic idea of the matrix factorization approach was already described, a hardware realization to prove the idea with real numbers was not given yet. Furthermore, [16] and [15] suggested a horizontal decomposition of the matrix $\mathbf{W}$. With respect to hardware realization, the vertical decomposition proposed in (2), is much better suited as we will show later on. Therefore, with this paper we first introduce a hardware realization, based on reconfigurable logic (FPGAs), which was dedicatedly designed for this approach. We will show, that using this new approach and our hardware architecture, we can save up to 80% hardware resources compared to a standard design flow on FPGAs.

The underlying matrices of many DSP algorithms, e.g. Fourier transforms, are fixed and application-independent. Thus, it is simple to design an architecture implementing them. For the case of ANNs, these matrices differ from application to application and layer to layer. But due to the fact, that weight matrices are created only once for an application, but are reused for every inference, we can utilize the reconfiguration ability of FPGAs to address any ANN. Moreover, the internal structure of the CC matrices can be perfectly utilized by FPGAs, since shift-operations are just wiring on an FPGA, which will cost neither additional hardware resources nor energy. Implementations on application-specific integrated circuits (ASICs) also benefit from the latter point but lack the aspect of reconfigurability. This means, FPGAs will be the perfect candidate for this kind of algorithm. In this paper we show the combination of *matrix decomposition and reconfigurable logic* for the first time. In Figure 1, this concept is shown graphically and compared to the state-of-the-art (SoA) solution using an ANN as example: At the left, an example of a weight matrix is shown. For general CMMs, as they are used in, e.g., DSP algorithms, the approach stays the same, only the matrix differs. The traditional approach to a matrix-vector-product architecture requires many multipliers and adders. In contrast, our approach presented later in Section IV benefits from the well-behaved structure of the CC-matrices

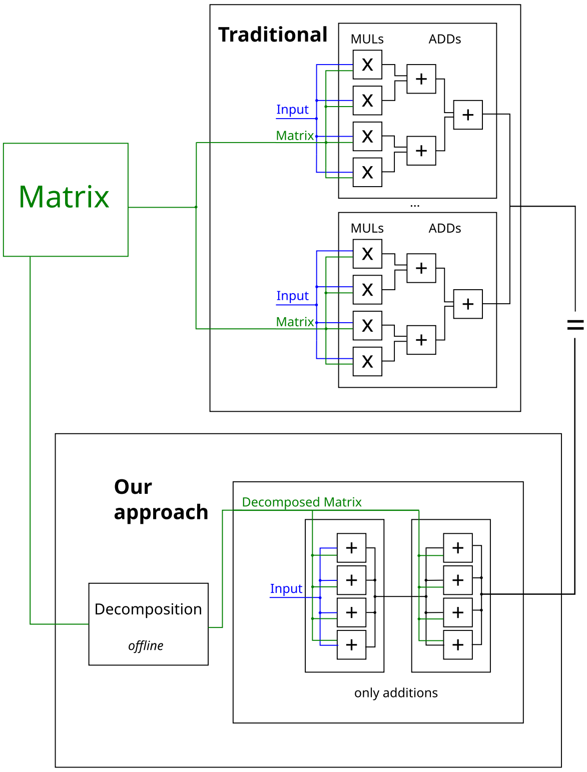


FIGURE 1. **Comparison between state-of-the art mapping of ANN (at the top) and linear computation coding (at the bottom) onto reconfigurable hardware.**

and does only require shifters and a fixed small amount of adders. Additionally, the linear computation coding approach decomposes the original matrix into multiple CC-matrices. Due to their unique structure, a resource-aware hardware mapping is possible, which results in limited usage of adders and a short critical path.

This paper is structured as follows. In the introduction, we show the importance of this topic and explain the basic idea. Section II discusses previous related work in two domains, first hardware architecture approaches and second developments from an algorithmic point of view. Further, in Section III we present our computation coding approach of decomposition of matrices in a detailed way. Afterwards, Section IV explains the architecture and hardware realization of our approach on reconfigurable hardware, first for general CMMs and later for ANNs. In Section V, we prove the working principle of our architecture by explaining our experiments and evaluating their results. Additionally, Section VI provides a further implementation of a multi layer perceptron (MLP) with further efficiency comparisons to other implementation methods. Finally, Section VII concludes the paper.

Throughout the paper, matrices and vectors are denoted by boldface upper case and boldface lower case letters, respectively. Non-bold indexed letters denote the entries of the respective matrices and vectors in boldface. Design variables are denoted by non-bold upper case letters. Lower case non-bold letters denote indices running from 0 or 1 to the respective upper case letter.

II. RELATED WORK

A. HARDWARE ARCHITECTURES

One of the main drivers of deep learning was the vast amount of computational resources graphic processing units (GPUs) could provide to train and evaluate sufficiently powerful neural networks. However, with the widespread usage of deep learning and the expansion to further domains like automotive, mobile, and edge devices, additional factors like energy efficiency, latency, and runtime predictability became more urgent. For this reason, a substantial amount of research has focused on the acceleration of neural networks with specialized hardware in the last years [17]. Hereby, three main directions of optimization can be found in literature, which are not mutually exclusive, but are often combined for even greater benefits.

The first category is the design of data-driven digital circuits and its automation. While GPUs with their single-instruction multiple-threads-(SIMT)-style architecture offer many computational units with less control logic than central processing units (CPUs), they are still fully programmable. Hence, they inherently have a considerable amount of overhead, which is not needed for the smaller subset of operations in deep learning. Therefore, specialized dataflow architectures came in the focus of interest. One of the first candidates for this purpose were systolic arrays, which were already concisely described in 1978 [18]. Their locally connected structure of processing elements not only reduces the control hardware, but also increases the amount of local data movement. As a consequence of the fewer slow external memory accesses, this approach also mitigates the widening processor-memory gap, which has the potential to considerably improve performance and energy consumption. Due to these benefits, the concept has been used in many current designs and most prominently in Google's Tensor Processing Unit (TPU) [19], [20], [21]. For the same reasons, dataflow processing schemes have been similarly applied in varying scales to other architectures [22]. On a small scale, GPUs nowadays also incorporate specialized cores that efficiently process 4 × 4 matrix-matrix multiplications [23]. Furthermore, coarse-grained reconfigurable arrays (CGRAs) have been employed as a trade-off between programmability and efficiency [24], [25]. Hereby, the programmable processing cores directly source data from and provide data to other nearby cores via a routing fabric to keep data as local as possible. In the other extreme, several approaches propose to entirely forgo control flow and generate dedicated accelerators for specific networks [14], [26]. These architectures usually map layers or the complete model to own hardware for the highest efficiency at the cost of flexibility. While automation frameworks for all kinds of deep learning accelerators are nowadays indispensable, in particular these latter types make heavy use of network metadata like the number ranges of input, intermediate, and output values or the composition of the weights matrices [27], [28].

Due to the direct influence of the network structure on the efficiency of the accelerator circuits, optimizations usually already begin at the network itself. In this second direction of optimization two main approaches have emerged in literature. First, the quantization of weights and data from 32 bit floating point to a fixed point representation with a smaller bit width [29], [30]. This method has two benefits. It reduces the complexity of arithmetic operations while at the same time decreasing the amount of memory needed for weights. Therefore, a single operation is not only more memory efficient, but more can be calculated at once with the same memory bandwidth. As smaller bitwidths can also be found in other application domains, traditional architectures of CPUs and GPUs already incorporate vector processing capabilities. However, these are usually limited to fixed sizes of 8 bit, 16 bit, 32 bit and 64 bit. Despite the recent support of further operand types like `int4` and `bfloat16`, the optimal values can heavily vary between neural networks and do often not coincide with these fixed widths. Therefore, several approaches use hardware that is specifically adapted for the applications by quantizing the network as far as ternary or binary weights [14], [26], [28]. Adjacent to the quantization, pruning has been established as the second way to prepare a network for optimized hardware [31]. Here, weights are successively set to zero and then stored in compressed formats. Although this method makes the control flow logic more complex to parse the weight storage, the overall amount of arithmetic operations is drastically reduced as multiplications and additions with 0 can be completely stripped away. This leads to a sparse matrix multiplication, which can be calculated faster and with less energy than the original [32], [33]. Further research has explored the optimization of constant matrix multiplication by converting entries to the CSD representation and then optimizing the resulting adder tree for the matrix-vector multiplication [12], [13]. While some approaches discuss finding an optimal exact solution to the matrix vector multiplication [12], there are also efforts to reduce accuracy for further reduction in hardware cost of the resulting designs [13].

While such dataflow architectures and their network optimizations are also the main focus of this paper, they can be further combined with technology driven designs. This third main direction of research extensively utilizes unconventional or novel circuitry and memory cells. As such, one of the central structures are crossbar arrays, which usually follow the general principle of dataflow architectures. They internally store the network weights and perform analog multiplications and additions as the information medium propagates through them [34], [35], [36]. Hereby, a number of different technologies with their own benefits and drawbacks have been investigated. On the still rather conventional side are designs based on capacitors [37] and common non-volatile memory cells like flash and silicon-oxide-nitride-oxide-silicon (SONOS) [38], which are already used in traditional circuits and are therefore more reliable. Regarding novel components, memristive memory cells have become a field of active research for deep learning [34], [35], [36], [39]. As non-volatile, electrically alterable resistances, they enable storage and in-memory computing in the same device. Furthermore, they promise a high cell density and simpler fabrication in conjunction with digital logic cells due to the full complementary metal-oxide-semiconductor (CMOS) compatibility [40]. Aside from the classical data processing with electric circuits, silicon photonics also has been presented as an approach for deep learning [41], [42]. Due to its unprecedented possible bandwidth, photonic computing systems promise high performance and energy efficiency. However, there is still a long way until these systems are industrially viable outside of the network communication sector [43]. Although, our approach presented in this paper is based on classical electrical circuits, it can be combined with these technology-driven optimizations in the future.

B. ALGORITHMIC FUNDAMENTALS

From the pioneering work of Strassen [44] and improvements of the same [45], we know that matrix multiplication can be performed more efficiently than by the standard method of calculating inner products of rows and columns. However, the Strassen algorithm brings only benefits for matrix ranks in the thousands and beyond. Furthermore, applying Strassen's ideas to ANNs requires buffering the input vectors until an input matrix with sufficiently large rank has been accumulated. Thus, the Strassen algorithm and its further improvements have remained a well-studied subject in theoretical computer science, but not entered algorithm design for matrix-vector multiplication in ANNs. In this work, we follow a very different line of ideas, instead.

Higher accuracy of computation, in general, results in higher computational load. Any improvement in the former is thus equivalent to a reduction of the latter. Both are two sides of the same tapestry, which is utilized in the sequel.

The common way to represent matrices is to element-wise quantize their entries. The more accurate the quantization of each entry, the more accurate is the whole matrix. The entries are typically quantized by the common signed integer representation. Each additional binary digit halves the average quantization error. This can be improved by Booth's CSD representation [46]. Each CSD reduces the average root mean-square quantization error by a factor $\sqrt{28}$ [16].

When implementing small CMMs, as they appear in, e.g., DSP algorithms, the CSD representation brings further benefits. Instead of implementing full multiplication units, we can convert the sum of products (SOP), that represents the computation of one line of the CMM, into a directed acyclic graph (DAG) of adders which then can be minimized by reusing intermediate results where possible [12]. As this problem is NP-hard [12], finding good solutions for large matrices, as they appear, e.g., in ANNs, is not viable. In more recent work, inaccurate implementations are considered, trading accuracy for even lower hardware costs [13].

The element-wise CSD representation is simple, but leaves much room for improvement. The coordinate rotation digital computer (CORDIC) algorithm [47] represents 2×2 matrices as products of 2×2 matrix factors that only contain signed powers of two and is used to improve the calculation of, e.g., trigonometric functions. Recent work on linear computation coding in [15] shows that rectangular matrices are much better suited to be decomposed into matrix products than square matrices. Furthermore, the savings grow unboundedly with matrix size. This behavior was first observed for the particular example of the mailman algorithm [48]. While the latter is too inflexible for practical applications, modern methods of linear computation coding work well for matrices of almost any size and aimed accuracy of computation. The particular algorithm utilized in this work is detailed in the sequel.

III. OUR METHOD: COMPUTATION CODING - DECOMPOSITION OF MATRICES

Our objective is to decompose a matrix $\mathbf{W}$ in such a way that the product $\mathbf{W}\boldsymbol{v}$ can be computed with minimum effort on an FGPA.

The multiplicative decomposition algorithm in [15] works much better for rectangular than for square matrices. Therefore, we first slice the matrix $\mathbf{W}$ into S tall sub-matrices $\mathbf{W}_s$ as in (2). Similarly, the vector $\boldsymbol{v}$ is cut into S sub-vectors $\boldsymbol{v}_s$ such that $\boldsymbol{v}^\dagger = [\boldsymbol{v}_1^\dagger | \boldsymbol{v}_2^\dagger | \dots | \boldsymbol{v}_S^\dagger]$. Thus, we have

$$\mathbf{W}\boldsymbol{v} = \sum_{s=1}^{S} \mathbf{W}_s \boldsymbol{v}_s. \tag{4}$$

Note that reference [15] slices the matrix $\mathbf{W}$ into wide, not tall sub-matrices. This requires the subsequent factorization algorithm to operate on the transposed matrices. Although horizontal slicing results in a similar number of required computations, it is less suited for pipelining: Vertical slicing ensures that all computation paths have exactly the same lengths, cf. equal number of nonzero entries in the rows in (7). Horizontal slicing, however, results in varying lengths of computation paths, cf. equal number of nonzero entries in the columns in (7). With vertical slicing we ensure minimal clock skew in hardware implementation. All paths have the same lengths and thus a minimal clock skew is guaranteed. Horizontal slicing on the other hand leads to varying paths lengths and thus the clock skew increases. It is well established that an optimized clock skew is key to designing efficient hardware [49].

Each tall sub-matrix $\mathbf{W}_s$ is decomposed into P nontrivial matrix factors $\mathbf{F}_{s,p}$ as denoted in (3). For this purpose, we use a recursive approach to be detailed in the sequel. The recursive approach is not optimal and more sophisticated decompositions may yield even better results. However, it performs well and allows for a matrix decomposition with reasonable complexity.

We initialize the recursion with the trivial factor $\mathbf{F}_{s,0} = [\mathbf{I}|\mathbf{0}]^\dagger$ with $\mathbf{I}$ and $\mathbf{0}$ denoting the identity and the all-zero matrix, respectively. The sizes of the matrices $\mathbf{I}$ and $\mathbf{0}$ are chosen such that $\mathbf{F}_{s,0}$ and $\mathbf{W}_s$ have the same size. This initialization works well for most weight matrices occurring, in practice. However, it may perform poor in some exceptional cases, e.g., for matrices that contain only positive or only negative entries. In that case other initializations should be taken, see [15] for details.

We calculate the matrix factor $\mathbf{F}_{s,p}$ given the previous matrix factors $\mathbf{F}_{s,p-1}$ and the sub-matrix $\mathbf{W}_s$. With M denoting the number of rows in $\mathbf{W}_s$, $p > 0$, and some parameter E, we solve

$$\boldsymbol{f}_{s,p,m} = \underset{\boldsymbol{\varphi} \in \{0, \pm 2^{\mathbb{Z}}\}^M : \|\boldsymbol{\varphi}\|_0 = E}{\operatorname{argmin}} \left\| \boldsymbol{w}_{s,m} - \boldsymbol{\varphi} \mathbf{F}_{s,p-1} \cdots \mathbf{F}_{s,0} \right\|_2 \tag{5}$$

row-wise for all rows $\boldsymbol{f}_{s,p,m}$ of $\mathbf{F}_{s,p}$. There $\boldsymbol{w}_{s,m}$ and $\|\boldsymbol{\varphi}\|_0$ denote the m-th row of $\mathbf{W}_s$ and the number of non-zero components in $\boldsymbol{\varphi}$, respectively. We stop the recursion if the desired accuracy is reached, i.e. the Frobenius norm of the difference between the approximation and the exact weight matrix is small enough. Thus, the desired accuracy determines the number of non-trivial factors P. While the initial and trivial factor $\mathbf{F}_{s,0}$ is rectangular having the same size as $\mathbf{W}_s$, all subsequent factors $\mathbf{F}_{s,1}$ to $\mathbf{F}_{s,P}$ are square.

The optimization problem (5) is NP-hard. Therefore, we resort to an approximate solution based on a quantized version of matching pursuit [50]. First, we find the first non-zero entry of the vector $\boldsymbol{\varphi}$. For that purpose, we calculate all matchings and quantize their scale factors to the most suitable signed powers of two. Then, we pick the best matching with respect to the Euclidean distance to the vector $\boldsymbol{w}_{s,m}$. Given this first entry of $\boldsymbol{\varphi}$, we find the second entry of $\boldsymbol{\varphi}$. We repeat that, until E non-zero entries are found.

An example of such a decomposition is given in the sequel. Consider the matrix[1]

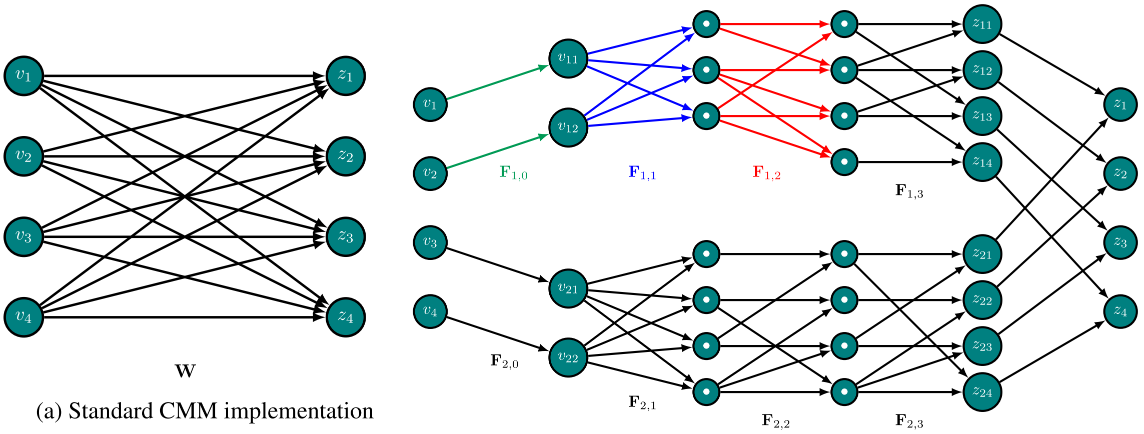
$$\mathbf{W}_1 = \begin{bmatrix} 0.5377 & 0.3188 \\ 1.8339 & -1.3077 \\ -2.2588 & -0.4336 \\ 0.8622 & 0.3426 \end{bmatrix}. \tag{6}$$

For $P = 2$ and $E = 2$, we approximate it as

$$\mathbf{W}_1 \approx \underbrace{\begin{bmatrix} 1 & 0 & -\frac{1}{32} & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & -\frac{1}{8} & 1 & 0 \\ 0 & -\frac{1}{16} & -\frac{1}{2} & 0 \end{bmatrix}}_{\mathbf{F}_{1,2}} \underbrace{\begin{bmatrix} \frac{1}{2} & \frac{1}{4} \\ 2 & -1 \\ -2 & -\frac{1}{2} \\ 1 & \frac{1}{4} \end{bmatrix}}_{\mathbf{F}_{1,1}\mathbf{F}_{1,0}}. \tag{7}$$

In order to approximate the matrix-vector product $\mathbf{W}_1\boldsymbol{v}_1$, we first calculate the vector $\mathbf{F}_{1,1}\mathbf{F}_{1,0}\boldsymbol{v}_1$ which requires four additions, and subsequently multiply this vector by $\mathbf{F}_{s,2}$ which also requires four additions, so eight additions in total.

[1] This matrix is found when executing the command `A = randn(4,2)` right after starting Matlab. It is chosen to demonstrate that this example is not made up particularly to promote this algorithm, but it is a generic example.



(a) Standard CMM implementation

(b) Computation Coding CMM implementation

FIGURE 2. Comparison of data dependencies of the standard implementation and the computation coded computation of a CMM using a 4 × 4 matrix. The colored data paths $\mathbf{F}_{1,0}$ to $\mathbf{F}_{1,2}$ of the CC implementation are also presented in Equation 7.

The signal-to-quantization noise ratio

$$\text{SQNR} = \frac{\|\mathbf{W}_1\|_\text{F}^2}{\|\mathbf{W}_1 - \mathbf{F}_{1,2}\mathbf{F}_{1,1}\mathbf{F}_{1,0}\|_\text{F}^2} \tag{8}$$

for this example is given by 24 dB which corresponds to the accuracy of 4-bit signed-integer arithmetic. The SQNR measured in decibels was found in [15] to scale linearly with the number of factors P, so any desired accuracy can be reached. Note that a direct computation of $\mathbf{W}_1\boldsymbol{v}_1$, irrespective of the accuracy, would require four additions and eight multiplications.

By design, any matrix factor $\mathbf{F}_{s,p}$, $p > 0$ contains exactly E nonzero elements per row. These E non-zero elements are signed powers of two. Multiplying such a matrix to a vector, thus, requires at most E shifts and exactly $E - 1$ additions (or subtractions) per row. For an $M \times N$ weight matrix, these are $M(E - 1)$ additions (or subtractions) for any matrix factor $\mathbf{F}_{s,p}$. In total, there are PS of these matrix factors. Moreover, we have $(S-1)M$ additions for calculating the sum in (4). Thus, the total number of additions and subtractions to compute $\mathbf{W}\boldsymbol{v}$ is

$$(E - 1)MPS + (S - 1)M. \tag{9}$$

The choices of the three parameters P, S, and E determine both the computational effort and the accuracy of the approximation (3). Setting

$$S \approx N/\log_2 M \tag{10}$$

is typically not a bad choice. The optimum value of S often deviates from (10) by at most a factor of two in one or the other direction. For given parameter S, the parameters P and E are chosen such as to reach the desired accuracy of computation.

In the standard approach of computing CMM, every output is a direct linear combination of any input. This is visualized in Figure 2a, where every node represents a vector entry and the edges connecting them depict the data dependencies. Each entry of the resulting output vector is the sum of products of input vector entries and the matrix entries in the corresponding column. Therefore, all entries of the result vector directly depend on all entries of the input vector. The structure in Figure 2a is generic for all 4 × 4 matrices that do not contain zeros. The particular properties of the constant matrix are encoded in the coefficients of the linear combinations at the output nodes and are not visible in Figure 2a.

The computation coded decomposition of the same matrix is visualized in Figure 2b for $P = 3$ factors. No data dependencies are lost by applying the proposed decomposition. Instead, previously direct data dependencies are exchanged with indirect dependencies. The result of the CMM is not computed directly from sums of products, but by repeated CMMs with CC-matrices for each slice of the original matrix, followed by accumulation of all slice approximations. The 4 × 4 matrix $\mathbf{W}$ of the CMM is sliced into $S = 2$ slices of size 4 × 2. Decomposition of the first slice $\mathbf{W}_1$ is presented in Equation 7 for $P = 2$ factors. Each slice computation is now only dependent on the respective two elements of the input vector. After $P = 3$ factors with one addition each ($E = 2$), the slice-wise computation is finished and the final accumulation takes place.

Between the first and second matrix factor, a node appears to be missing. Instead of four nodes, there are only three. A one-to-one translation of (7) would actually make this fourth node to show up. However, due to the all-zero column in $\mathbf{F}_{1,2}$, this node has not any outgoing edges. Thus, it does not have any influence on the final result and can be eliminated. Optimizing VDHL compilers remove such nodes automatically. See also [51] for details.

The two data dependency graphs mainly differ in two points:

- For CC, the structure of the graph depends on constant matrix $\mathbf{W}$, while for the standard approach, it does not.

CC encodes the information about the matrix $\mathbf{W}$ predominantly in the structure of the graph and only to a minor extent in the weights of the linear combinations at the nodes.
- For CC, all nodes have a fixed number of incoming edges which can be freely chosen by the design variables E and S. For the standard approach, however, the number of incoming edges is equal to the number of rows of the matrix $\mathbf{W}$.

Our approach allows, due to choice of the parameters E and S, to design the number of incoming edges to computation nodes freely. Thus, we reduce the node activity from four edges to two in our example. This reduction in node activity is much more pronounced for larger matrices. For the purpose of readability we chose to present this small 4 × 4 example.

The matrix decomposition described above is not the only sensible method of linear computation coding. A recent alternative requiring even less additions is reported in [51]. Whether the method in [51] is also well suited for implementation on FPGAs is to be explored in future work.

IV. OUR METHOD: ARCHITECTURE AND HARDWARE-REALIZATION

In this section, we propose an architecture for implementing CC-matrix-vector products. It utilizes the particular properties of the CC-matrices. Results of several experiments on the scalability of our architecture are presented and further aspects needed for the real-world implementation are elaborated upon.

Our objective is to design an optimized architecture implementing MLPs, as a general form of ANNs, that can be realized on FPGAs. A MLP is a sequence of neural layers, each layer consisting of a set of neurons with activation functions. The resulting activations of a layer can be computed elementwise or, when represented as a vector, using a matrix-vector product concatenated with a non-linear activation function as shown in (1). There, $\boldsymbol{a}$ is the resulting activation of the current layer with weight matrix $\mathbf{W}$, input $\boldsymbol{v}$, bias $\boldsymbol{b}$, and activation function ϕ. The inputs to a layer are the activations of the previous layer or, in the case of the first layer, the input to the MLP itself. Disregarding the activation function, it is immediately obvious that the matrix-vector product is the most computationally expensive component of (1). Thus, when designing an optimized MLP architecture, it is crucial to focus on said multiplication. This coincides with the implementation for general CMMs, as the ANN design consists, next to other parts, of CMM units. Our approach replaces the original CMM with multiple CC-matrix-vector products, or in other words CMMs where the underlying matrices are CC-matrices and are created using the approximate matrix decomposition algorithm presented in Section III.

A. ARCHITECTURE

This section will present our architecture starting with explaining the components of a CMM and discussing benefits stemming from certain restrictions to them.

The standard implementation of a CMM consists of two steps, the multiplication itself and the column-wise accumulation per entry of the result vector. Consider the product

$$\boldsymbol{z} = \mathbf{W}\boldsymbol{v} \tag{11}$$

where $\mathbf{W} \in \mathbb{R}^{M \times N}$, $\boldsymbol{v} \in \mathbb{R}^{N}$ and $\boldsymbol{z} \in \mathbb{R}^{M}$. When implementing the product (11) in a naive architecture, the computation can be separated into two distinct steps, 1) the multiplications themselves and 2) row-wise accumulations. Thus, we can define the intermediate matrix $\mathbf{W}' \in \mathbb{R}^{M \times N}$ whose rows $\boldsymbol{w}'_m = \boldsymbol{w}_m \odot \boldsymbol{v}$ are the element-wise (Hadamard) products of the rows of $\mathbf{W}$ with the input vector $\boldsymbol{v}$. As explained, now we need to row-wise accumulate the matrix $\mathbf{W}'$ to compute the resulting vector $\boldsymbol{z}$ with $z_m = \sum_{n=1}^{N} W'_{m,n}$. As already alluded to, we want to modify the product (11) to simplify the hardware required to implement it. Instead of using the original matrix $\mathbf{W}$, we make use of the approximate matrix decomposition algorithm presented in Section III. This results in the approximation of $\mathbf{W}$ such that

$$\mathbf{W}\boldsymbol{v} \approx \sum_{s=1}^{S} \prod_{p=0}^{P} \mathbf{F}_{s,p}\boldsymbol{v} \tag{12}$$

where $\mathbf{F}_{s,p} \in \mathbb{R}^{M \times M}$ for $p > 0$. There are a few parameters that determine the number of matrix-vector products needed to implement this decomposition. The algorithm decomposes $\mathbf{W}$ into slices of width W, as shown in (13).

$$W = \frac{N}{S} \tag{13}$$

Thus, with increasing width W the number of slices decreases. The parameters P and E are used to control the accuracy of the approximate decomposition which increases with P and E meaning that more factors and less sparsity in these factors yield a more precise result. Typically we want to set P and E such that we perform (at least) equally accurate as the integer-arithmetic used by the naive implementation. Each of the matrices $\mathbf{F}_{s,p}$ is a CC-matrix with the following properties that can be controlled by the algorithm:

- There is a fixed number of elements that are unequal to zero in each row of the matrix.
- The domain of values that matrix entries can be is fixed to a finite set.

The proposed architecture which is depicted in Fig. 3 benefits from both points mentioned and the following paragraph explains how both constraints are exploited.

We restrict each row of the matrix to consist of exactly two non-zero elements, i.e. $E = 2$. As each element of the output vector z_m is calculated as the inner product of two vectors with one of them containing only two non-zero entries, we only need one addition to compute z_m. This holds for any of the M components of $\boldsymbol{z}$, so there are M additions needed in total for this step. When implementing a general matrix vector product one needs to choose between a linear adder and a tree adder effectively choosing between minimizing hardware cost and critical path length. To implement a matrix vector product

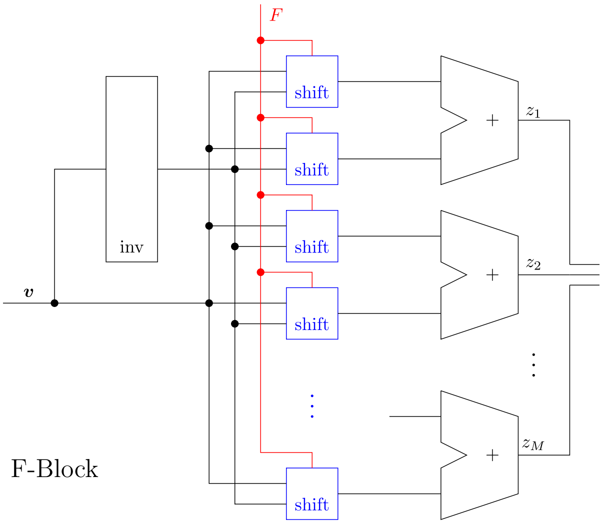


FIGURE 3. Architecture of CC-matrix-vector product (F-Block) $Fv = z$ with input v (left) and output z (right). Additional multiplications with -1 are taken care of by the inverter-module denoted by *inv*. The blue shifters can have varying implementations while the black part is fixed. Depending on the type of implementation of the blue part, the red network is needed or not. In our implementation the blue and red parts are replaced by hard-wired shifts.

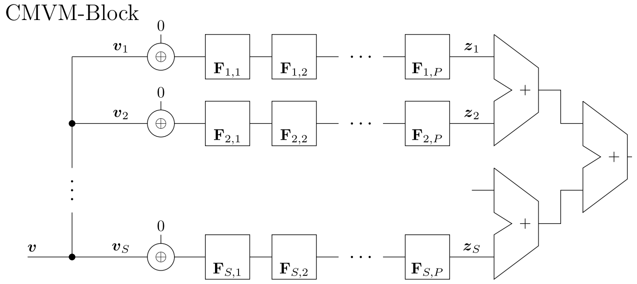


FIGURE 4. Architecture of approximate matrix-vector product (CMVM-Block) $\mathbf{W}v = z$ where $\mathbf{W}$ is decomposed into the CC-matrices $\mathbf{F}_{1,1}$ to $\mathbf{F}_{S,P}$. The input for $\mathbf{F}_{s,1}$ is the s-th part of v separated into S slices and zeros such that the vector has N elements. After parallel computation, the partial results z_S are accumulated to z.

with the described restriction we only need one adder per matrix row optimizing both hardware cost and critical path length at the same time.

As a side note, with an increase in E the number of adders required to accumulate the intermediate results per row may increase. The optimization problem here is between minimizing hardware cost by choosing a linear adder structure or minimizing the critical path by choosing tree adders. While E drives hardware cost per CC-matrix product, the total hardware cost is balanced out by the need of less sequential products. Due to more information being stored in each CC-matrix the number P of CC-matrices required to reach a certain precision decreases.

The main benefit of our approach compared to a naive implementation results from the second bullet point mentioned above. By restricting all non-zero matrix entries to be signed powers of two, we need not any multiplication elements to implement the matrix-vector product. As numbers are encoded binary, a multiplication with a power of two is nothing but a shift. There are various possibilities to implement these shifts. Barrel shifters enable shifting in both directions and thus are one way of implementing the required computation. The main benefit of this approach is that the implementation is independent of matrix values as matrix elements are the controlling input of the shifters and can be read from memory. When assuming the matrices as fixed, we can skip the shifters and hard-wire the shifts using simple connections between the input vector and the adders.

At last, as we do not restrict the matrices to consist of positive values only, we need a way to handle negative matrix entries. This is done by inverting the input vector at the beginning and choosing between the inverted and the original input vector at the time of shifting. From an overall perspective an implementation of a CC-matrix-vector product compared to a naive implementation of a general product has a significantly lower hardware cost and critical path length.

As was pointed out, architecture in Fig. 3 only implements the CMM for CC-matrices. To implement a full product we need to assemble multiple instances of the mentioned architecture as shown in Fig. 4. The architecture can be divided into three sections, construction of input vectors, multiplication with CC-matrices and accumulation of partial results.

The construction of the input vectors is needed as a first step, because each row of CC-matrix-vector products only approximates a slice of the original matrix. Thus, we only need the corresponding section of the input vector $\boldsymbol{v}$. To match the dimensions of the matrices $\mathbf{F}_{s,p} \in \mathbb{R}^{M\times M}$ for $p > 0$, the partial input vector gets multiplied with an identity matrix augmented by zeros. This is formally done in (3) by the initial matrix factor $\mathbf{F}_{s,0}$. This can be shortened to filling up the remaining bits with zeros. This is done in the leftmost section of Fig. 4.

After having assembled the partial input vectors, an array of CC-matrix-vector products follows. Each of these implements the architecture presented previously. Each row is implemented as a chain of products running in parallel to other rows.

As each row of products only represents a subset of columns of the original weight matrix, or the underlying matrix of general CMMs, the results of a row of CC-matrix-vector-products is only a partial result. To get the final output vector all partial results z_s need to be accumulated which is best done in a binary tree structure. This approach minimizes the critical path length at the cost of more hardware to implement it when compared to a linear addition.

As was explained in Section III, the decomposition of a matrix via the CC-algorithm is only approximate. The more consecutive factors there are per slice, the higher the accuracy of the approximation [15]. To achieve viability compared to other, competing implementations of CMMs we simply use as many factors in the decomposition to reach the same or better accuracy which the fixed-point arithmetic of the competing implementation would provide. When we want to approximate, e.g., 8 bit signed integer arithmetic in this way, we need to set the amount of computations per slice such that the quantization error is 48 dB below the entries of the weight matrix [52].

Our designs are implemented using the Very-High-Speed Integrated Circuit Hardware Description Language (VHDL) which is generated from the output of the decomposition algorithm using a hardware generator we implemented in `python`. This choice best bridges the semantic gap between the output and corresponding interface of the decomposition algorithm and the hardware descriptions required for synthesis while only relying on basic assertions and operations supported by most VHDL standards and also comes with the benefit of providing an interface for common neural net frameworks such as `Tensorflow` or `PyTorch`.

The goal of our implementation is a tool that generates the description of the instantiated designs as shown above while sustaining compatibility with most synthesis tools, not only for FPGA implementations but for ASIC implementations as well. Due to this, we implement our designs in the VHDL-93 standard [53] which is supported by most synthesis tools. This choice comes with the drawback that the VHDL-93 standard lacks features of more modern versions which makes implementations based on it unnecessarily complex and hard to read. Therefore we have chosen to implement a hardware generator instead of relying on static VHDL implementations of our designs. As mentioned our generator is implemented in `python` and consists of a generic VHDL generator backend and several functions using the backend to generate the descriptions of the designs for specified input matrices and parameters. Thus, the resulting interface consists of python functions which can either be called individually but can also be connected to neural net interfaces as mentioned previously. In-between the decomposition algorithm is executed, this also takes place on the `python`-level.

This architecture implements an approximate CMM, with the approximation being at least as accurate as comparable fixed-point arithmetic. The resource-efficiency we achieve is not at the cost of a lower throughput. It arises from suitably quantizing matrices rather than naively quantizing their entries. Therefore, it can replace the naive implementation of CMMs without hindering accuracy or throughput. In the following the potential of the presented architecture will be explored with the main focus on properties of weight matrices of ANNs.

B. SCALABILITY

There are several factors that affect the scalability of our architecture for a matrix-vector product. Apart from optimizations to the architecture and the ease of applying them, we can also explore the effects of variable matrix traits. The latter will be explored in the upcoming two experiments which consider the impact of matrix dimensions as well as the statistics of matrix entries. After that we present our approach to pipelining the architecture demonstrating how we can make use of the repetitive architecture and optimize critical paths.

A particular problem is the well-known memory bottleneck, i.e. to enable our architecture to compute fast we require high data throughput. A matrix with dimensions 64×64 already requires as input-output-(IO)-ports two vectors with 64 entries resulting, when encoded in 8 bit, 1024 bit transferred every clock cycle. At a frequency of 400 MHz we need a memory bandwidth of 400 Gbit/s. To solve this requirement we chose to implement our designs for the XCVU37-ES1 chip by Xilinx on the ADM-PCIE-9H7 board by Alpha Delta. This setup is consistent for all the following results.

Our design is a fully rolled-out implementation and thus executes the corresponding CMMs in one clock-cycle. Therefore, we compare our architecture to a fully rolled-out version of the naive implementation. As a basis for comparison of hardware cost we implement both the standard approach and our CC-approach using look-up-tables (LUTs) and do not use any DSPs present on this specific FPGA. By doing this we can guarantee a fair comparison in terms of the validity of the results as well as the applicability to other FPGA boards.

Floating-point computations introduce a level of accuracy which can be used as a termination criterion for the decomposition of matrices, if desired. This way, similar to how we achieve the accuracy of fixed point computations, the approximative decomposition provided by the CC-algorithm can then be as accurate as floating point arithmetic.

Our designs are individual dataflow architectures that implement an entire CMM, ANN layer, or ANN, respectively. There is no need for mapping algorithms for processing elements. Results concerning speedup can be found in the experiment concerning pipelining in Section IV-B3, the main experiment which discusses timing results, as well as in the last implementation realizing a MLP in Section VI.

1) MATRIX DIMENSIONS

One key aspect of the performance of our architecture is its scalability in terms of varying matrix dimensions and the corresponding benefit when compared to the naive implementation. This facet will be explored in the following experiment. As we want to represent matrices appearing in ANNs we chose to test our approach on square matrices with dimensions ranging from 64×64 to 256×256. To keep generality we randomly generated matrices with independent uniformly distributed entries. An experiment on varying statistics of entries is presented in Section IV-B2.

The main choices left before running the linear computation coding algorithm is the precision we want to achieve and the size of the matrix slices to approximate. We compare our results to a fixed-integer arithmetic naive implementation of a matrix-vector product with a bit width of 8 bit. The bitwidth of all vector entries between matrices, meaning the in- and outgoing vectors of the corresponding matrices, is set to 8 bit. This determines the precision we need to achieve. According

TABLE 1. ***S*: Number of vertical slices per matrix, *P*: number of consecutive products per matrix slice, *W*: width of each slice. The standard approach (STD) implements a naive matrix-vector product with fixed-point arithmetic with a bit width of 8. The precision of said bit width is achieved by the computation coding (CC) decomposition resulting in *S* consecutive products per slice. The column $I = \frac{\text{STD LUTs}}{\text{CC LUTs}}$ represents the improvement of our approach over the naive one.**

Dimension $M \times N$	W	S	P	CC LUTs	STD LUTs	I
32 × 32	2	16	6	13089	27888	2.1 x
32 × 32	4	8	8	12128	27888	2.3 x
32 × 32	8	4	21	16579	27888	1.7 x
64 × 64	2	32	6	38525	115761	3.0 x
64 × 64	4	16	6	34027	115761	3.4 x
64 × 64	8	8	13	40474	115761	2.9 x
128 × 128	2	64	3	134600	421967	3.1 x
128 × 128	4	32	6	133510	421967	3.2 x
128 × 128	8	16	11	138548	421967	3.0 x
256 × 256	2	128	3	449165	1471861	3.3 x
256 × 256	4	64	5	444498	1471861	3.3 x
256 × 256	8	32	10	461538	1471861	3.2 x

to (10) and (13), the slice width should be around 5 to 8. We chose the three options $W = 2$, $W = 4$, and $W = 8$, as they lead to numbers of slices which are powers of two, thus simplifying the adder trees in Figure 4.

The results of this experiment are presented in Table 1. Note that there are no multiplication units and all addition units are implemented as LUTs. It is immediately obvious that our approach outperforms the standard implementation in every case. The factor by which our implementation is better in terms of hardware cost measured in LUTs required for implementation ranges from 2.3 to 3.4 for the best slice width shown in the table. The amount of adders required to implement our approach depends on the matrix dimension and the precision we want to achieve. Counting multipliers as multiple adders, we can expect a theoretical factor of $\frac{1}{2}\log_2 M$ for the benefit in terms of the number of adders of our approach compared to a naive implementation for an $M \times N$ matrix [15]. This theoretical factor is also approximately reflected in our results, where we count LUTs instead of adders. Further we learn that the precise slice width used in the decomposition of the original matrix has only minor impact. The number of consecutive CC-products P required to achieve the desired precision decreases with the number of slices S. This compensates for the increased hardware cost stemming from a larger number of slices.

The theoretical improvement factor $\frac{1}{2} \log_2 M$ grows with matrix size. For slice width $W = 8$, this is confirmed by the LUTs count in Table 1. The optimum theoretical slice width, however, is not a power of two, in general, but may be of course, in particular cases. This can explain why one result in Table 1 sticks out with particularly excellent performance, i.e. $M = N = 64$ and $W = 4$. Here, the optimum slice width seems to be close to $W = 4$, while it does not match the grid $W \in \{2, 4, 8\}$ for larger matrices.

Our approach outperforms a naive implementation of a M-by-N-matrix-vector product by a factor close to $\frac{1}{2}\log_2 M$ mainly depending on M, the number of rows of the matrix. Note that this factor is not dependent on the number of columns of the matrix due to the slicing applied to it. By slicing the matrix we achieve that the number of rows dominates the number of columns, e.g. in Table 1 by a factor of 16, 32 or 64, resulting in the observed relation between implementation cost and matrix rows. Had we sliced the weight matrix horizontally instead of vertically, the factor would be $\frac{1}{2}\log_2 N$ depending on the number of columns instead of rows.

2) MATRIX ENTRY STATISTICS

In the previous experiment we considered matrices with varying dimensions but all of the corresponding matrices have uniformly distributed entries. The overall goal of our research is to achieve a fair and well-founded comparison between our CC-approach and other competing implementation methods. To be able to achieve such a comparison with our results we need to point out that in the real world not all matrices feature a uniform entry distribution.

There are well known optimizations for special kinds of multiplications. Consider the product shown in 14 where x is supposed to be a fixed value.

$$z = xv \tag{14}$$

To implement 14 in hardware, generally a multiplication unit consisting of several adders is needed. The same is not true for special values of x.

- $x = 0$: There is no multiplication needed, the result of 14 is $z = 0$.
- $x = 1$: There is no multiplication needed, the result of Equation 14 is $z = v$.
- $x = 2^y$, $y \in \mathbb{Z}$: As numbers in hardware are represented in the binary system, there is no multiplication needed. The result can be computed by shifting v by y digits.
- $x = \sum_{y=i}^{j} 2^y$, $i, j \in \mathbb{Z}$: This is the representation of a binary number consisting of 0-bits and one continuous sequence of 1-bits. Using a general multiplication $j - i$ values have to be accumulated to calculate Equation 14. The CSD representation provides the optimization $x = 2^{j+1} - 2^i$. For sequences of more than two 1-bits the amount of additions needed can be reduced to just one. This comes with no added hardware cost, as subtractions in hardware can be realized by addition units with no extra expense. Such a multiplication unit is called a *Booth*-Multiplier.

Concluding the list of optimizations, the actual matrix entries in a CMM have a large influence on the implementation cost of the entire CMM. This observation is the motivation for the following experiment. We explore the impact of varying 0-1-bit ratios on the improvement that the CC-approach provides over the standard implementation of CMMs.

The uniform distribution that was used in the previous experiment is now represented by a 50% 0-bit matrix. Such a uniform distribution is the worst case for multiplication, as it features the highest number of additions required for implementation while also providing the least amount of Booth-Multipliers. Small 0-bit ratios feature more Booth-Multipliers

TABLE 2. This data compares the hardware complexity of a naive implementation of a matrix-vector product compared to our computation coding approach for matrices with varying percentage of 0-bits. Each matrix has the dimension 64 × 64 and is encoded in 8-bit to compute the metric. The factor $I = \frac{\text{STD LUTs}}{\text{CC LUTs}}$ is the improvement of our computation coding approach over the naive implementation. Each matrix is sliced with a slice width of $W = 4$ resulting in $S = 16$ slices each. The number of factors is displayed in column P.

0-bits	P	CC LUTs	STD LUTs	I
0%	2	9753	976	0.10 x
5%	4	18999	76012	4.00 x
10%	4	19068	86353	4.52 x
20%	5	26289	93947	3.57 x
30%	6	33266	104955	3.16 x
40%	6	34468	108457	3.15 x
50%	6	34662	114422	3.30 x
60%	6	33980	106395	3.13 x
70%	6	34825	94664	2.72 x
80%	5	28254	66087	2.34 x
90%	4	21014	36143	1.72 x

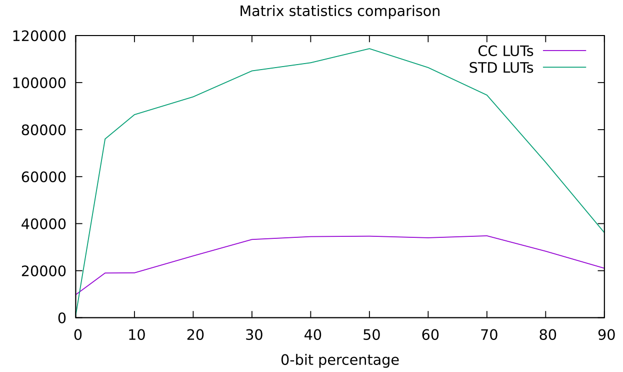


FIGURE 5. This plot visualizes the results presented in Table 2.

in the standard implementation while higher 0-bit ratios require less additions overall. All matrices have a fixed dimension of 64 × 64 and are sliced with a slice-width of $W = 4$. Again, the accuracy that is used to determine the termination criterion of the decomposition algorithm is the quantization error of 8 bit fixed-point arithmetic (−47 dB). The matrices are generated starting with the zero-matrix by distributing a certain number of 1-bits uniformly over all binary representations of entries. Table 2 presents the comparison of hardware cost of the CC-approach versus the standard implementation for matrices with various 0-bit ratios.

As expected the implementation of a nonuniform matrix is not as expensive as one for a uniform matrix which is true for both the naive approach being marked as STD in Table 2 as well as our architecture. These results are graphically presented in Figure 5. We can also see that in general our approach is better by a factor of 3 to 4.5 compared to the naive implementation with some anomalies on the edge cases. In the case of a matrix only consisting of 1-bits we see that the naive implementation is actually better. This is due to the matrix decomposition only being approximate and the naive implementation making use of the static pattern of the matrix. For matrices consisting only of 0-bits or only of 1-bits the standard approach also does not require multiplication units.

The improvement provided by the CC-approach over the standard implementation does not drastically decrease with deviations from a uniform entry distribution. It is even larger for moderately small percentages of 0-bits. Only for extreme bit ratios, e.g. ≥ 80% or < 5% 0-bits we see a notable reduction in the improvement factor compared to the uniform case. This demonstrates that the CC-approach leads to a great improvement over the standard implementation for a wide range of entry statistics. Uniformly distributed entries are not required for a large improvement by our method.

Performance of the CC-approach is not hindered by minor deviations from the uniform distribution. A sufficiently balanced bit distribution leads to a clear improvement of about three times over the standard implementation. For a not too extreme bias towards 1-bits, the improvement can even be larger.

3) PIPELINING

There are various approaches to implement a pipeline into the architecture seen above. The traditional approach is to pipeline the architecture top-down. This means to insert pipeline-registers between each CC-matrix-vector product, further between each matrix-vector product, eventually between the various computational steps in each layer, and between the layers themselves. An abstract illustration of such a hierarchical pipelining approach is presented in Figure 6. Hierarchical pipelining without further synchronization is possible as each row of multipliers has the same number of elements and thus every path through said multipliers has the same length.

As alternative to the hierarchical approach it is also possible to implement a bottom-up approach. For bottom-up pipelining we consider the architecture as an entirely rolled out net with adders as base-building blocks and partition it into subnets with equal critical path lengths. It is possible to use adders as the atomic units of this process because our proposed designs for CMMs is made up of adders only and does not rely on multiplication units. As all paths through the architecture have the same length and thus every path being the critical path, synchronization between the different paths is not necessary.

It is possible to create pipeline steps not only between CC-matrix-vector multiplications but also inside the computation units implementing said products themselves without the need for additional synchronization. As for the implementation of each individual CC-product, the critical path length remains constant for each multiplication. This is due to each matrix row requiring the same number of multiplications with elements of the input vector and each individual multiplication being realized as a shift only. With these static properties there is little variance in path length over all paths in the implementation of a CC-matrix-vector product.

The only difference between the hierarchical and the net-partitioning approach to pipelining is the amount of

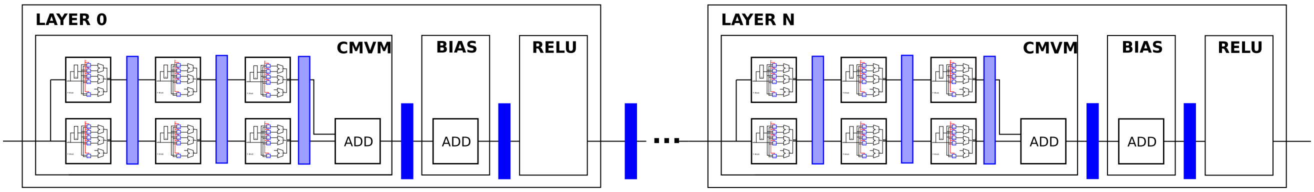


FIGURE 6. This figure shows an abstract approach to pipelining which is being implemented in our architecture. Each register is depicted in blue. A pipeline step spans a CC-matrix-vector product, a bias addition or a nonlinear activation function (e.g. here RELU). The CMVM units represent a CC-matrix-vector product each making use of our optimized approach.

registers required. Depending on the placement of a pipeline step the number of signals that need to be buffered varies. A set of registers placed as a pre- or postfix to a CMVM unit requires only the corresponding input or output vectors to be stored. When we cut a CMVM unit itself in partitions, the pipeline registers in-between the stages have to store the intermediate vectors of all concerned slice approximation datapaths.

Our approach to pipelining sees the multiplication as an unfolded net and simply inserts pipeline steps such that the critical path of each step has the same length. In the case of a fixed matrix this benefits highly from the architecture only being made up of adders, as shifts can be hard-wired. Therefore, an optimal pipeline distribution becomes possible and can even be computed beforehand. To explore the effects of pipelining in our architecture we compare randomly generated matrices for uniformly distributed entries with various counts of pipeline steps each.

Next to the resulting hardware complexity for each product the most important results are the corresponding frequencies that the implementations can be run at. Said maximal frequency is determined by the critical path length, the longest run of gates between two registers. To determine the optimal frequency we make use of the bisection method. For each implementation run of our architecture we set a fixed timing goal. After the implementation we determine the difference in timing between the goal and the required time for the critical path. According to the gathered information we adjust the timing goal until the absolute difference passes a termination threshold giving us the maximal frequency of the corresponding design.

The test procedure was repeated for a set of amounts of pipeline steps for a 64 × 64 matrix with two respective approximate decompositions. For all our results the width of the vector entries is set to 8 bit. Each decomposition requires a different amount of concatenated CC-products per row of computation to reach 8-bit integer calculation precision.

The results of this experiment are presented in Table 3 where several observations can be made. First of all, the hardware cost increases with the increasing number of pipeline steps where the LUT counts required for implementation are about constant but the number or required registers increases.

TABLE 3. This data compares the number of LUTs, the number of flip-flops (FFs), and the maximal frequencies for various designs for the decompositions of a 64 × 64 matrix with uniformly distributed entries. Each frequency is found using the bisection method starting with 100 MHz. The decomposed matrix approximates the original up to an error similar to fixed-point 8-bit arithmetic of a naive approach.

Pipeline steps	W	S	P	LUTs	FFs	MHz
1	4	16	6	35653	1077	99
8	4	16	6	35204	32486	370
1	8	8	13	38934	1030	90
14	8	8	13	37485	31592	377

The amount of additional registers per added pipeline step depends on the positioning of the step. While registers in-between layers or generally outside of the matrix-vector product result in a small increase of the register count, having pipeline steps inside the multiplication unit is more expensive. This is due to the parallel rows of computation which require to put registers in every row. Still both types of pipeline steps lead to a linear increase in required registers.

With an increase in pipeline steps the maximal frequencies of the according designs increase, reaching a peak at about 400 MHz. The maximal frequency is the same for implementations requiring more sequential CC-matrix-vector products as for implementations with fewer ones, as the minimal pipeline steps only depend on the greatest atomic units in the chain which are adders in both cases. The only difference in the resulting implementations for the two cases are the number of pipeline steps.

Note that with an increase in pipeline steps the initiation period of the overall pipeline also increases by the same amount of clock cycles. Table 3 show increases from one to eight and 14 pipeline steps respectively. The initiation period of the corresponding implementation also increases to eight and 14 cycles. These clock cycles are shorter then the clock cycles of the design without pipeline steps, reducing the effects of said impact. After the initiation of the pipeline the architecture is back to the single-cycle execution of the corresponding F-Blocks.

As is described in Section IV-A, our designs are generated implementations using our own `python` VHDL generator and do not rely on existing high-level synthesis (HLS) tools. Thus existing loop initiation algorithms and procedures are not applicable to our architecture. A high number of pipeline steps does not harm the efficiency as there are no hazards occurring during computation. Overall these improvements gained by introducing pipelining to our designs lead to a speedup of 3.7 and 4.2 respectively. With initiation periods being directly related to the number of pipeline steps, pipelining the architecture only leads to an improvement when more then a single calculation is performed. The more vectors are passed through the hardware, the less the initiation cycles impact the total number of cycles required for the overall computation. Reconfiguring the FPGA to accommodate instances of a partitioned implementation can cut this pipelining improvement. Therefore, it is best to implement the fully rolled-out net as a whole or buffer input data.

As is shown in Section IV-B1, the required amount of sequential factors to achieve a certain accuracy is not only dependent on the slice width but also on the matrix dimensions. The results in Table 1 show that for a fixed slice width said number only varies slightly while for varying slice widths it changes drastically. Therefore, we can conclude that the results shown in Table 3 are similar for matrices with larger dimensions. Even if more or less sequential factors are required to achieve the desired accuracy, only the number of initiation steps for the matrix changes, but the maximal clock frequency does only vary marginally. This is also reflected in the similar maximal frequencies of the two explored designs in Table 3.

4) GENERAL COMPARISON WITH OTHER CSD-ALGORITHMS

As was already alluded to earlier, there are algorithms aiming to lower the computational effort and thereby hardware cost of the corresponding implementations for CMMs, in particular [12], [13]. The general approach of said algorithms is to convert matrix entries to CSD and thereby minimize the number of non-zero bits appearing in the matrices [12]. The resulting additions of a SOP in a line of the CMVM unit is then represented as a DAG of adders which then can be minimized.

Said minimization problem is NP-hard [12]. Especially for matrices with large dimensions, as are, e.g., used in our benchmarks earlier, it is not feasible to find the optimal solution to this minimization problem. Thus, greedy searches are used to approach the optimum [12] or inaccuracy is introduced to the calculation [13]. This results in an improvement of computational complexity of the CMM which in turn is reflected in more hardware resource-efficient implementations. The results presented by Kumm et al. [12] feature an improvement of up to 34% while the results presented by Aksoy and Flores [13] similarly reach an improvement of up to 30%.

Our proposed method reaches improvement factors of three to five times, or in other words saves 67% to 80% hardware cost while not hindering throughput. In summary, our linear computation coding approach produces better results than current state-of-the-art (SoA) algorithms.

V. EVALUATION USING AN EXAMPLE DLRM

For the purpose of analyzing our architecture, we chose to use a recommender system as example of an ANN. These systems are utilized by various companies, e.g., for streaming services to give their customers advice about movies they may like based on their consumer behavior. During the last years this systems have become increasingly reliable in their forecasts not least because of the more frequent use of algorithmic models aided by multilayer perceptron (MLP) concepts. One of these algorithms was implemented recently in 2019 by the Deep Learning Recommendation Model for Personalization and Recommendation Systems (DLRM) [54].

A. PRINCIPLES OF RECOMMENDATION NETWORKS

In order to better understand the value of this model's single components, we first give a short introduction on the principles of recommendation networks. Recommendations today are given based on two underlying principles namely content-based filtering and collaborative filtering. While the former approach bases its prediction on the users' own preferences, collaborative filtering tries to imply a solution based on the preferences of similar users. One of the first systems making advantage of both of these concepts was the factorization machine. Its prediction formula consists of two parts, the regressive and the matrix factorization one. The regression handles both sparse and dense data of the feature vector and can consequently be seen as the content-based filtering part of the system. The matrix factorization on the other hand accounts for the interactions between feature blocks, which represents the collaborative filtering approach. Even though both of these models are already integrated in this straight forward implementation, results can be further refined by making use of MLP layers. Due to its non-linearity it is possible for MLPs to learn even higher degrees of interactions.

DLRM now brings those ideas together and introduces a new concept by separating the features into dense continuous and sparse categorical features, which are represented by embedding vectors of the same size. The dense features are then fed into a bottom MLP which transforms them into an intermediate vector of the same size as the embedding vectors of the categorical features before. Similar to the factorization machine in the second stage, now the dot product between the embedding vectors and the output of the bottom MLP is computed, which represents the computation of second-order interactions of different features. The products are then concatenated to the result from the bottom MLP and fed into another top MLP and finally to a sigmoid function in order to obtain a probability.

In order to test our approach, we exchanged the weights in the MLP layers of an already trained DLRM network with the ones obtained by the utilization of our matrix decomposition algorithm.

TABLE 4. Comparison of the hardware costs of implementing the matrix-vector products of each layer of the DLRM. In this table, *Layer* denotes the name of the corresponding layer, with an $M \times N$ weight-matrix. S is the number of slices with width $W = 4$ and P the number of consecutive CC-matrix-products. $I = \frac{\text{CC-LUTs}}{\text{STD-LUTs}}$ is the factor by which our approach improves the standard (naive) implementation.

Layer	M	N	S	P	CC-LUTs	STD-LUTs	I
Top 1	256	512	64	5	527518	1806426	3.4 x
Bot 1	512	256	128	5	1052470	2029566	1.9 x
Bot 2	256	64	16	6	44769	277837	6.2 x
Bot 3	64	16	4	10	4872	24340	5.0 x
Total					1629629	4138169	2.5 x

B. COMPARISON

As a basis for comparison, we chose the same hardware platform as for all other experiments presented above. This means we synthesized our design for the XCVU37-ES1 chip by Xilinx on the ADM-PCIE-9H7 board by Alpha Delta. First we look at a layer-by-layer comparison of our approach and a naive implementation of a trained ANN as described previously. The results are displayed in Table 4. Again it is immediately obvious that our approach performs better than the naive implementation with the improvement factor varying between 2x and 6x. It is notable that in the Bottom-2 layer said factor is very high compared to other results. This is due to properties of the matrix used in this layer. With the underlying matrix being a 64 × 256 matrix it is quite big compared to, e.g., the next layer only featuring a 16×64 layer. On top of that the matrix is not at all sparse leading to an overall high improvement over the naive implementation. The Bottom-1 layer features an even larger 256 × 512 matrix, but it is not as dense as the matrix of the Bottom-2 layer. Thus, the improvement of 1.9 x of using our approach compared to a naive implementation is not as high. Overall both the naive implementation and our approach require an enormous amount of LUTs to be implemented on a FPGA, but summing up all layers our approach saves 60 % of the hardware cost. As mentioned before, pipelining the resulting architecture is very efficient for our approach as the registers can be placed in a way that all paths through the pipeline step have the same length. This cannot be said for the naive implementation as a comparable assurance cannot be made.

VI. COMPARISON WITH AN EXAMPLE MLP

Previous examples explored the layer by layer performance of the CC-approach in comparison to a standard implementation of CMMs on the premise of single fixed matrices. This is also true for the previously presented DLRM which is explored on a layer-by-layer basis as the high amount of LUTs required to implement it as a whole is to high for FPGAs. With this final example we present a MLP which can be placed and implemented on a FPGA. Similar to previous research [55], we also choose to design a MLP to classify the Modified National Institute of Standards and Technology (MNIST) dataset.

An abstract representation of our net design can be seen in Figure 7. Our net consists of four layers with the corresponding weight matrices being of size 784×64, 64×64, 64 × 64, and, 64 × 10, respectively, each followed by tanh-activation functions. To accommodate the 28 × 28 greyscale input images of handwritten digits we reshape the input to a vector of dimension 784. The resulting classification is achieved by sorting the images into ten categories, one for each digit, hence the initial and final dimensions of the weight matrices of the layers of the net. With this setup, a learning rate of 0.001, and a batch size of 32, an average reliability of classification between 90 % and 97 % can be achieved after 30 generations.

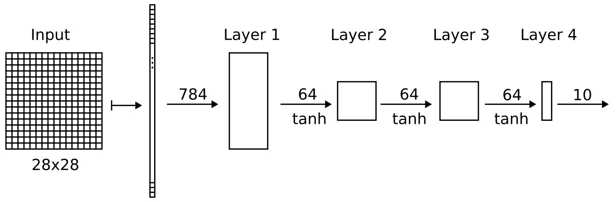


FIGURE 7. This figure shows an abstract representation of a MLP design used to classify the MNIST dataset. It reshapes the input images and passes them through four fully connected layers the dimensions of which can be found in the image.

TABLE 5. Test accuracy of the net presented in Figure 7 after 50 epochs of training with different methods of computation. Compared are the standard floating point method included with pytorch, an 8 bit fixed point implementation and the corresponding CC-decomposition. The target accuracy of the decomposition is set to 48dB, such that the approximation is equally or more accurate than the 8 bit fixed point calculation.

Method	Test Accuracy
Floating Point	94.03%
Fixed Point (8bit)	91.96%
CC (SQNR $\geq$ 48 dB)	92.43%

We trained a net with these parameters and achieved an accuracy saturation at 94 % in the test-dataset after 30 epochs. Fixed point calculation methods, as well as the CC-decomposition approximating it, introduce an additional quantization error. The influence of the this quantization error is small, as shown in Table 5. Table 5 shows a comparison between the default floating point computation method provided by the PyTorch framework, as well as a fixed point implementation and a corresponding approximation using the Computation Coding algorithm presented in Section III. The results show only a minor decrease in reliability of classification by the net when changing the computation method. Note that not only the quantization error, but also the additional inaccuracy introduced by using our decomposition compared to the standard floating point approach is lower than for the fixed point implementation.

Similar to previous examples we use the weight matrices of the layers as the basis for CMMs which then are implemented using our approach and a standard approach. As target accuracy the bitwidth of 8 bit is chosen and again the same board from Alpha Delta based on the ADM-PCIE-9H7 chip by Xilinx is used as platform for implementation. For the decomposition of the matrices we tried various different slice

TABLE 6. This table shows the hardware cost in LUTs for our CC-approach compared to the standard implementation. The decomposition arguments are the slice width W and the corresponding slice count S as well as the amount of factors P used. The last row shows the total cost including implementations of tanh-activation functions, while the other rows present layer-by-layer results of the individual weight matrix CMMs. $I = \frac{\text{STD LUTs}}{\text{CC LUTs}}$ describes the improvement of our approach over the standard implementation.

	Dimension				LUTs	LUTs	
Layer	$M \times N$	W	S	P	CC	STD	I
1	64×784	4	16	5	329029	771727	2.3 x
2	64×64	4	16	6	31738	84749	2.7 x
3	64×64	4	16	6	33343	87997	2.6 x
4	10×64	2	5	3	4846	16736	3.45 x
Total					412108	970905	2.4 x

widths of $W = 2$, $W = 4$ and $W = 8$, the number of factors for each decomposition is chosen to match the desired accuracy of 47 dB.

The results of this experiment with the optimal configuration of slice widths for the decomposition on a layer-by-layer basis can be seen in Table 6. According to our findings the optimal slice widths for the decomposition of the weight matrices of the first three layers is $W = 4$ while the last weight matrix is sliced in five slices of width $W = 2$. Note that the first, as well as the last layer have extreme dimensions in the sense that their corresponding horizontal and vertical dimensions differ a lot. Therefore, to achieve better results these matrices are not sliced horizontally, as opposed to the weight matrices of the remaining two layers, yielding more tall and narrow matrix slices. In the layer-by-layer analysis of the implementation costs of the second and third layer we improve by a factor of 2.7 x. Previous results, i.e. Table 2, suggest that the improvement is not as high as suggested by Table 1 due to non-uniformly distributed matrix entries. The first layer shows that our approach also decreases hardware cost for matrices with one extreme dimension while the last layer shows that, when slicing is done correctly, matrices with one small dimension can also be implemented in efficient CMMs.

For the implementation of the entire net, the weight matrices are concatenated with adders for the bias as well as the implementation of the non-linear activation function, in our case a tanh-function. With the chosen configuration we achieve an overall improvement of 2.4 x which can be seen in the last row of Table 6. The numbers are the results of the implementation of the entire net including the application of tanh activation functions in between the multiplications with the corresponding weight matrices. With the final configuration and thus the hardware designs fixed we now can compare power and timing results between the standard implementation, our CC-approach as well as a CPU and GPU execution of the inference of the neural net. We found the target frequency of the overall net to be at 372 MHz with 39 pipeline stages, coinciding with the results presented in Table 3. Vertical slicing of the weight matrices in the first and last layer leads to varying path lengths. There are paths that would require more than one addition per row and thus there is an extra pipeline step introduced. Paths where this problem does not occur require an additional buffer stage in the corresponding pipeline stages. Additional hardware cost is introduced by this necessary buffering. Still this decomposition requires less LUTs then the implementation of the corresponding vertically sliced decomposition due to the extreme dimension-ratio of the underlying matrices.

TABLE 7. Dynamic power analysis of the CC-approach and the standard implementation. The frequency is fixed at 50 MHz. The analysis is run with a toggle rate of 12.5% and a static probability of 0.5. Column CC displays the dynamic power draw of the CC-approach, column STD the same of the standard implementation. I denotes the improvement.

Layer	CC [W]	STD [W]	I
Layer 1	0.034	0.062	1.82 x
Layer 2	0.035	0.064	1.83 x

Eventually we now also introduce the energy aspect. For a power comparison between the standard implementation and our architecture we setup a layer-wise comparison between the two methods. With a fixed throughput we enable a fair comparison between the two combinatorial designs, i.e. both designs are not pipelined. The last point of comparison which is missing is the comparison of power requirements between the two implementations. We ran a power analysis based on subnet switching activity provided by a post-synthesis simulation the results of which can be seen in Table 7. The frequency is fixed to 50 MHz to generate results for equal through-puts. Further parameters are kept default with a toggle rate of 12.5% and a static probability of 0.5. Following this setup we compared the implementations of layers two and three of our MNIST-MLP presented in Figure 7. Layer two of Figure 7 shows an improvement of 1.82 x, while the implementations of layer three feature an improvement of 1.83 x. These findings show that our design not only provides a reduction on the required number of LUTs for the implementation of these CMMs and thus the entire net, but are more energy efficient than the standard implementation of the same. One key aspect of our future research is further analysis of these results in combination with ASIC implementation comparisons of the respective designs.

To achieve a comparison to the execution of inference of this net to the performance of a CPU we ran inference with 2000 classifications. Our test system is equipped with two AMD EPYC ROME 7352 CPUs featuring 48 cores clocked at 3.2 GHz and a NVIDIA A100 (40GB) GPU. The measurement was set up by copying a random image from the MNIST dataset as an input to the corresponding memory, after that, inference of the net described above was repeatedly executed and the execution times were measured. To achieve reliable results the process was repeated ten times and the measured times were averaged. Also the number of times any individual image was used as an input without any more storage accesses was varied between 10 and 100000 times to guarantee a saturated execution time per inference. In our measurement our CPU executed an average of 1554 inferences per second while our GPU achieved 5688 inferences per second. In comparison our hardware-design implemented on the ADM-PCIE-9H7 achieves a target frequency of 378 MHz while not saturating the available memory bandwidth of the HBM memory modules. A design with register only between the layers can still be run with 50 MHz while keeping the number of required registers to a minimum. Compared to the execution of the net on the mentioned GPU and CPU the CC-solution still provides a notable speedup of 32175 x over the CPU and 8790 x over the GPU implementation. Note that this speedup is possible with an overall power budget of under 10 W while the NVIDIA A100 has a TDP of 300 W and the AMD EPYC ROME 7352 CPUs come with a TDP of 155 W each. The power draw of these devices thus is not comparable to the small power budget of our FPGA solution. With these findings we can infer that our approach to the execution of CMMs on FPGAs does not only outperform standard implementations of this computational operation but also provides an enormous improvement in performance over SoA solutions such as CPU and GPU implementations.

VII. DISCUSSION AND CONCLUSION

In this paper, we presented a new method for lowering the computational effort of CMMs, e.g., for ANN inference, decomposing the constant (weight) matrices by slicing and factorization. The resulting sub-matrices are sparse, with a well-behaved structure and contain only numbers related to a power of two. Utilizing this a-priori knowledge, an efficient computer architecture is designed, which exploits the structure of the sub-matrices perfectly. Finally, hardware resources can be decreased by a factor of 2 to 6.

While in this work, the main focus is set on MLP ANNs, an increasing number of today's applications use convolutional neural nets (CNNs). We already found a method to apply the linear computation coding procedure to this kind of ANNs. Investigations are ongoing. Future work here includes a modified decomposition algorithm as well as hardware architecture to support CNNs.

Additionally, in this paper we focused on implementing an architecture for CMMs that are equally or more accurate than fixed-point implementations. Especially inference in ANNs do not always require this high level of accuracy. Thus, it is possible to lower computational accuracy of certain applications while still achieving similar results. Hence another future point of interest of ours is to explore the benefits of tuning down computational accuracy and thereby improving hardware efficiency even further. In this aspect we expect our architecture to do well as there are two approaches we can take here. First, it is possible to simply use less consecutive matrix factors to approximate each matrix-slice. Second, we can reduce the number of slices, which also reduces the accuracy of computation for a fixed number of matrix factors. Clearly, both approaches can be mixed in order to get a suitable trade-off between hardware-efficiency and accuracy of computation. In this direction of research comparisons to other computation paradigms, such as floating point arithmetics, are also of importance. As is already alluded to, the variable accuracy of the decomposition by the CC-algorithm can also be used to achieve the same accuracy that floating point arithmetics provide. Future work will explore the hardware cost of such a implementation and provide a comparison to existing floating point implementations, e.g. to dedicated DSPs as well as to LUT-based implementations.

Another point of focus in our future research will be experimenting with varying entry-counts per row of a CC-matrix. Instead of fixing the structure to only allow for two entries per row, it is also possible to use more powers of two. With only one entry there is no addition needed while four, eight or more entries similar to the traditional approach require larger adder implementations. With a higher number of entries not only the number of adders per CC-matrix vector product increases but also the number of matrix factors required to approximate the original matrix decreases. This relation will be explored further. Also different adder implementations like adder-trees and linear adders can be compared in different aspects like hardware cost and critical path length.

Building upon all the mentioned future research, we will explore ways to implement our designs beyond FPGAs. For implementations of DSP algorithms specialized accelerators already exist and our approach to CMM improves on them. In this aspect, we will explore ASIC implementations of our architecture. As already mentioned in the beginning, the general downside to ASICs when compared to FPGAs is the lack of reconfigurability. In these regards we will explore the performance of our design on CGRAs or even specialized reconfigurable ASIC-like implementations where only the interconnections, i.e. the wiring that replaces the shifters in the CMM units for CC-matrices, are reconfigured.

REFERENCES

[1] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[4] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a Gaussian Denoiser: Residual learning of deep CNN for image denoising," *IEEE Trans. Image Process.*, vol. 26, no. 7, pp. 3142–3155, Jul. 2017.

[5] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[6] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6645–6649.

[7] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Trans. Audio, Speech Language Process.*, vol. 22, no. 10, pp. 1533–1545, Oct. 2014.

[8] P. Bangalore and L. B. Tjernberg, "An artificial neural network approach for early fault detection of gearbox bearings," *IEEE Trans. Smart Grid*, vol. 6, no. 2, pp. 980–987, Mar. 2015.

[9] Y. Xu, Y. Sun, X. Liu, and Y. Zheng, "A digital-twin-assisted fault diagnosis using deep transfer learning," *IEEE Access*, vol. 7, pp. 19990–19999, 2019.

[10] O. Gustafsson, J. Coleman, A. Dempster, and M. Macleod, "Low-complexity hybrid form fir filters using matrix multiple constant multiplication," in *Conf. Rec. 38th Asilomar Conf. Signals, Syst. Comput.*, vol. 1, 2004, pp. 77–80.

[11] N. Boullis and A. Tisserand, "Some optimizations of hardware multiplication by constant matrices," *IEEE Trans. Comput.*, vol. 54, no. 10, pp. 1271–1282, Oct. 2005.

[12] M. Kumm, M. Hardieck, and P. Zipf, "Optimization of constant matrix multiplication with low power and high throughput," *IEEE Trans. Comput.*, vol. 66, no. 12, pp. 2072–2080, Dec. 2017.

[13] L. Aksoy, P. Flores, and J. Monteiro, "A novel method for the approximation of multiplierless constant matrix vector multiplication," *EURASIP J. Embedded Syst.*, vol. 2016, no. 1, pp. 1–11, Dec. 2016.

[14] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN- R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, pp. 1–23, Dec. 2018, doi: 10.1145/3242897.

[15] R. R. Müller, B. Gade, and A. Bereyhi, "Linear computation coding," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Toronto, ON, Canada, Jun. 2021, pp. 5065–5069.

[16] R. R. Müller, B. Gade, and A. Bereyhi, "Efficient matrix multiplication: The sparse power-of-2 factorization," in *Proc. Inf. Theory Appl. Workshop (ITA)*, San Diego, CA, USA, Feb. 2020, pp. 1–6.

[17] C. Latotzke and T. Gemmeke, "Efficiency versus accuracy: A review of design techniques for DNN hardware accelerators," *IEEE Access*, vol. 9, pp. 9785–9799, 2021.

[18] H. T. Kung and C. E. Leiserson, "Systolic arrays for (VLSI)," Dept. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-CS-79-103, 1978.

[19] 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, 2017, pp. 1–12, doi: 10.1145/3079856.3080246.

[20] L. Jia, L. Lu, X. Wei, and Y. Liang, "Generating systolic array accelerators with reusable blocks," *IEEE Micro*, vol. 40, no. 4, pp. 85–92, Jul. 2020.

[21] L. D. Medus, T. Iakymchuk, J. V. Frances-Villora, M. Bataller-Mompean, and A. Rosado-Munoz, "A novel systolic parallel hardware architecture for the FPGA acceleration of feedforward neural networks," *IEEE Access*, vol. 7, pp. 76084–76103, 2019.

[22] S. Kala, B. R. Jose, J. Mathew, and S. Nalesh, "High-performance CNN accelerator on FPGA using unified winograd-GEMM architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 12, pp. 2816–2828, Dec. 2019.

[23] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 522–531.

[24] K. Rocki, D. Van Essendelft, I. Sharapov, R. Schreiber, M. Morrison, V. Kibardin, A. Portnoy, J. F. Dietiker, M. Syamlal, and M. James, "Fast stencil-code computation on a wafer-scale processor," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–14.

[25] I. Bae, B. Harris, H. Min, and B. Egger, "Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2301–2310, Nov. 2018.

[26] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Learning FPGA configurations for highly efficient neural network inference," *IEEE Trans. Comput.*, vol. 69, no. 12, pp. 1795–1808, Dec. 2020.

[27] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, Jul. 2020, pp. 1–6.

[28] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.

[29] A. Demidovskij and E. Smirnov, "Effective post-training quantization of neural networks for inference on low power neural accelerator," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2020, pp. 1–7.

[30] A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jegou, and A. Joulin, "Training with quantization noise for extreme model compression," 2020, *arXiv:2004.07320*, doi: 10.48550/ARXIV.2004.07320.

[31] G. B. Hacene, V. Gripon, M. Arzel, N. Farrugia, and Y. Bengio, "Quantized guided pruning for efficient hardware implementations of deep neural networks," in *Proc. 18th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2020, pp. 206–209.

[32] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[33] T. Posewsky and D. Ziener, "A flexible FPGA-based inference architecture for pruned deep neural networks," in *Architecture of Computing Systems*. Cham, Switzerland: Springer, 2018, pp. 311–323.

[34] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-M. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, 2019, pp. 715–731, doi: 10.1145/3297858.3304049.

[35] R. Mochida, K. Kouno, Y. Hayata, M. Nakayama, T. Ono, H. Suwa, R. Yasuhara, K. Katayama, T. Mikawa, and Y. Gohou, "A 4M synapses integrated analog ReRAM based 66.5 TOPS/W neural-network processor with cell current controlled writing and flexible network architecture," in *Proc. IEEE Symp. VLSI Technol.*, Jun. 2018, pp. 175–176.

[36] O. Krestinskaya and A. P. James, "Binary weighted memristive analog deep neural network for near-sensor edge processing," in *Proc. IEEE 18th Int. Conf. Nanotechnol. (IEEE-NANO)*, Jul. 2018, pp. 1–4.

[37] Y. Li, S. Kim, X. Sun, P. Solomon, T. Gokmen, H. Tsai, S. Koswatta, Z. Ren, R. Mo, C. C. Yeh, W. Haensch, and E. Leobandung, "Capacitor-based cross-point array for analog neural network with record symmetry and linearity," in *Proc. IEEE Symp. VLSI Technol.*, Jun. 2018, pp. 25–26.

[38] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, and D. Fick, "Analog in-memory subthreshold deep neural network accelerator," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr. 2017, pp. 1–4.

[39] E. Rosenthal, S. Greshnikov, D. Soudry, and S. Kvatinsky, "A fully analog memristor-based neural network with online gradient training," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 1394–1397.

[40] (Jun. 2021). I. G. L.-I. für innovative Mikroelektronik. *IHP Offers Access to Memristive Technology for Edge AI Computing or Hardware Artificial Neural Networks Applications*. [Online]. Available: https://www.ihp-microelectronics.com/de/news/news-detailansicht/ihp-off%ers-access-to-memristive-technology-for-edge-ai-computing-or-hardware-artifici%al-neural-networks-applications

[41] M. A. Nahmias, T. F. de Lima, A. N. Tait, H.-T. Peng, B. J. Shastri, and P. R. Prucnal, "Photonic multiply-accumulate operations for neural networks," *IEEE J. Sel. Topics Quantum Electron.*, vol. 26, no. 1, pp. 1–18, Jan. 2020.

[42] V. Bangari, B. A. Marquez, H. Miller, A. N. Tait, M. A. Nahmias, T. F. de Lima, H.-T. Peng, P. R. Prucnal, and B. J. Shastri, "Digital electronics and analog photonics for convolutional neural networks (DEAP-CNNs)," *IEEE J. Sel. Topics Quantum Electron.*, vol. 26, no. 1, pp. 1–13, Jan. 2020.

[43] A. Rahim, T. Spuesens, R. Baets, and W. Bogaerts, "Open-access silicon photonics: Current status and emerging initiatives," *Proc. IEEE*, vol. 106, no. 12, pp. 2313–2330, Dec. 2018.

[44] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, 1969.

[45] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli, "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, no. 7930, pp. 47–53, Oct. 2022.

[46] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mech. Appl. Math.*, vol. 4, pp. 236–240, Jan. 1951.

[47] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electron. Comput.*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959.

[48] E. Liberty and S. W. Zucker, "The mailman algorithm: A note on matrix-vector multiplication," *Inf. Process. Lett.*, vol. 109, pp. 179–182, Jan. 2009.

[49] N. Maheshwari and S. S. Sapatnekar, *Clock Skew Optimization*. Boston, MA, USA: Springer, 1999, pp. 33–64, doi: 10.1007/978-1-4615-5637-4_3.

[50] S. G. Mallat and Z. Zhang, "Matching pursuit with time-frequency dictionaries," *IEEE Trans. Signal Process.*, vol. 41, no. 12, pp. 3397–3415, Dec. 1993.

[51] R. Müller, "Linear computation coding inspired by the Lempel-Ziv algorithm," in *Proc. IEEE Inf. Theory Workshop (ITW)*, Nov. 2022, pp. 606–611.

[52] R. M. Gray and D. L. Neuhoff, "Quantization," *IEEE Trans. Inf. Theory*, vol. 44, no. 6, pp. 2325–2383, Oct. 1998.

[53] (1993). I. S. C/DA. *IEEE 1076–1993*. Accessed: Oct. 17, 2022. [Online]. Available: https://standards.ieee.org/ieee/1076/1611/

[54] M. Naumov et al., "Deep learning recommendation model for personalization and recommendation systems," 2019, *arxiv:1906.00091*, doi: 10.48550/ARXIV.1906.00091.

[55] K. Khalil, A. Kumar, and M. Bayoumi, "Reconfigurable hardware design approach for economic neural network," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 12, pp. 5094–5098, Dec. 2022.

ALEXANDER LEHNERT received the master's degree in computer science from the Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2022. He is currently a Researcher at the Brandenburg University of Technology Cottbus-Senftenberg (BTU), Germany. His main research interest includes development and optimization of implementations of machine learning algorithms, with a focus on reconfigurable hardware.

PHILIPP HOLZINGER received the master's degree in computer science from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2017. He is currently a Researcher at the Chair of computer architecture, FAU. His research interest includes the design of heterogeneous system architectures, with a focus on reconfigurable and near-memory computing.

SIMON PFENNING received the master's degree in information and communication technology from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2019. He currently works as a Researcher at the Chair of computer architecture, FAU. His research interest includes the development and optimization of hardware platforms for machine learning.

RALF MÜLLER (Fellow, IEEE) received the Dipl.-Ing. and Dr.-Ing. (Hons.) degrees from Friedrich- Alexander-Universität (FAU) Erlangen-Nürnberg, in 1996 and 1999, respectively. From 2000 to 2004, he has directed a Research Group at the Telecommunications Research Center, Vienna, Austria, and taught as an Adjunct Professor at TU Wien. In 2005, he was appointment as a Full Professor at the Department of Electronics and Telecommunications, Norwegian University of Science and Technology, Trondheim, Norway. In 2013, he joined the Institute for Digital Communications at FAU in Erlangen, Germany. He was a co-recipient of the Leonard G. Abraham Prize from the IEEE Communications Society. He was presented awards for his dissertation by the Vodafone Foundation for Mobile Communications and the German Information Technology Society (ITG). He received the ITG Award for the paper "A Random Matrix Model for Communication via Antenna Arrays." He was also a co-recipient of the Philipp-Reis Award. He served as an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION THEORY, from 2003 to 2006, and an Executive Editor for the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS, from 2014 to 2016.

MARC REICHENBACH (Member, IEEE) received the Diploma degree in computer science from Friedrich-Schiller University Jena, Germany, in 2010, and the Ph.D. degree from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2017. From 2017 to 2021, he worked as a Postdoctoral Researcher at the Chair of computer architecture, FAU. Since 2021, he has been heading the Chair of computer engineering at the Brandenburg University of Technology Cottbus-Senftenberg (BTU), Germany, as a Substitute Professor. His research interests include novel computer architectures, memristive computing, and smart sensor architectures for varying application fields.