Efficient Mapping of CNNs onto Tightly Coupled Processor Arrays

Christian Heidorn*, Michael Witterauf, Frank Hannig, Jürgen Teich
Hardware/Software Co-Design, Friedrich-Alexander University Erlangen-Nürnberg, Germany.

* Corresponding author. Tel.: +49 9131 85 67312; email: Christian.Heidorn@fau.de
Manuscript submitted May 6, 2019; accepted July 25, 2019.
doi: 10.17706/jcp.14.8.541-556

Abstract: In this work, we show how to systematically map Convolutional Neural Networks (CNNs) onto Tightly Coupled Processor Arrays (TCPAs), a class of massively parallel accelerators for many computationally intensive tasks (e.g., from the digital signal and image processing domain). Contrary to previous approaches and implementations, we propose techniques for the layer-parallel execution of CNNs on processor arrays including the maximally overlapped processing of consecutive layers. This is achieved through layer fusion and loop unrolling to exploit the full pipelining potential of such massively parallel architectures for given CNNs. These transformations are also necessary to decrease the number of on-chip/off-chip data transfers. For CNNs, we present a calculus for achievable performance and memory requirements on TCPAs. Based on this calculus, it is shown how either throughput-maximal mappings can be determined for a given architecture. Alternatively, resource-minimized mappings to sustain a given throughput, e.g., number of frames per second, are systematically derived. The approach is evaluated for a CNN model for the MNIST benchmark on a processor array of size 4x4 including a comparison of the performance of the layer-parallel approach over layer-by-layer processing.

Key words: CNN accelerator, TCPA.

1. Introduction

Nowadays, CNNs (Convolutional Neural Networks) [1]-[3] are a primary approach in different computer vision tasks and are used for example in image recognition and image segmentation in images and videos [1]. The common machine-learning technique, where CNNs are utilized, is called supervised learning. Hereby, a CNN is trained for a specialized task on a dataset (i.e., a sequence of images). For the training phase, powerful systems, mostly GPUs and TPUs (Tensor Processing Units, [4]) have been proposed to accelerate the computationally expensive process [5]. For the inference part which describes, for example, if the trained network is used for classification of input data, often accelerators like ASICs, FPGAs, and CGRAs (Coarse Grained Reconfigurable Arrays) [5]-[11] are used, because they significantly decrease the needed energy and therefore are a great alternative for storage- and energy-limited embedded systems.

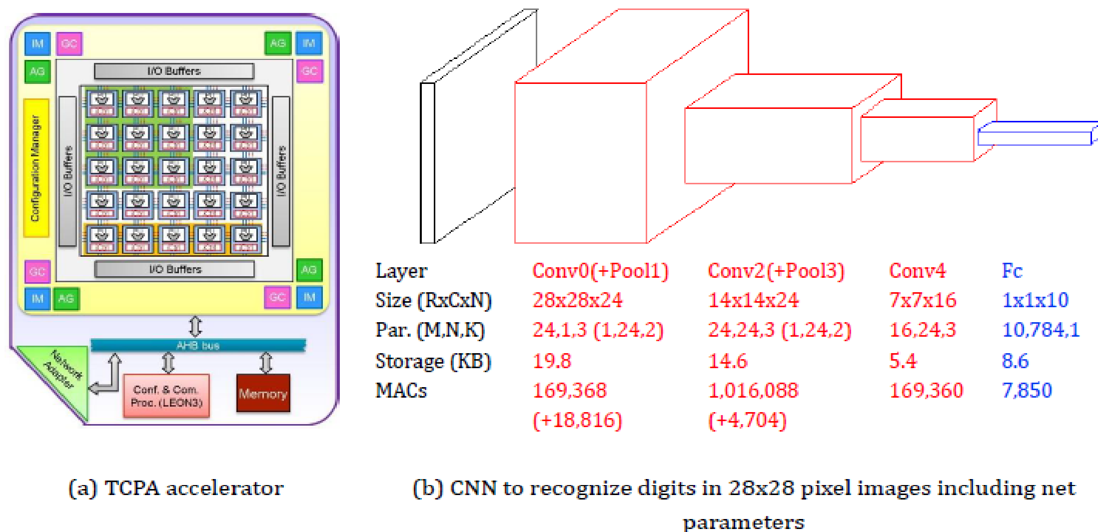
ASIC accelerators are designed for low energy devices like in [6], [7]. However, ASICs are not reconfigurable and are thus highly inflexible because they cannot be used to accelerate any other applications.

CGRA accelerators share features like a high number of parallel processing elements (PEs) and memory being located as near as possible to the PEs to decrease time and energy for data transfers [8]. One example is Tightly Coupled Processor Arrays (TCPAs) [12], [13], which belong to the class of accelerators that can be used in MPSoCs (Multi-Processor System-on-Chip) to accelerate loop programs in signal processing and

multi-media applications.

TCPAs are highly parameterizable architectures consisting of an array of tightly coupled lightweight VLIW (Very Long Instruction Word) PEs. Only PEs at the four borders have access to memory buffers to save energy (see Fig. 1(a)). Many parameters, such as the number of PEs, the number and type of functional units (i.e., branching, arithmetic, logic operators) within the PEs, and the connection between the PEs can be customized at synthesis time, offering the system designer a high degree of flexibility.

Belonging to the class of coarse-grain reconfigurable arrays (CGRAs), TCPAs offer a unique combination of multiple levels of parallelism, for example task-level parallelism, loop-level parallelism, iteration-level parallelism and instruction-level parallelism. TCPAs are thus particularly suited to accelerate computationally expensive nested loop programs that exhibit a high degree of parallelism such as CNNs. TCPAs can achieve a much higher energy-efficiency than general-purpose embedded processors or embedded GPUs [12]. For case studies including matrix-matrix multiplication, FIR-filtering, Gaussian blur filtering, and Harris corner detection, we refer to [12], [13].



(a) TCPA accelerator (b) CNN to recognize digits in 28x28 pixel images including net parameters

Fig. 1. (a) TCPA accelerator, such as can be integrated via a network adapter into an NoC (Network-on-Chip)-based MPSoC architecture. The abbreviation AG denotes address generators that are responsible for controlling the surrounding I/O buffers (several scratchpad memories at each border of the array). Also, depicted is a RISC processor, which primarily serves for communication tasks (i.e., scheduling of DMA transfers) and for configuring the accelerator. Further, it might be used for executing small tasks, such as the computation of one fully-connected (Fc) layer in our case study. (b) CNN to recognize letters and digits from 28x28 pixel images as used in our evaluation. The input (black rectangle on the left side) is a 28x28x1 grayscale image of the MNIST dataset [14]. The net consists of 3 convolutional layers (Conv0, Conv2, Conv4), 2 pooling layers (Pool1, Pool2) between two adjacent convolutional layers, and one fully-connected layer (Fc) for the classification into 10 classes. The table below shows the size of the feature map (rows (R) x columns (C) x number of features (N)) and the parameters (number filters (M), filter depth (N), kernel size (K)) for each layer (which will be described in Section 2.1). The storage requirements of each layer are deducted in Section 5.4.

Since convolutional layers of CNNs are based on matrix multiplications, they can be broken down into a 6-dimensional loop nest (see Section 2.2) which makes them well suited for TCPA acceleration. For retaining the flexibility of the architecture in accelerating different signal-processing algorithms, we provide methods for designing TCPA implementations of a given CNN such that the inference pass can be accelerated in a

highly efficient pipelining structure.

The main contributions of this paper may be summarized as follows:

- For the first time, it is shown that TCPAs are an excellent candidate for efficiently accelerating CNNs at a fixed throughput as well as maximizing throughput on a given architecture by providing a parallel and pipelined processing of multiple layers (*layer-parallel execution*).
- We utilize techniques, such as layer-parallel processing, loop unrolling and loop permutation, to best exploit the massive parallelism available in TCPA architectures.
- A performance and memory calculus is provided from which important mapping and scheduling decisions for the individual layers of a CNN can be deducted automatically.
- Finally, we evaluate our presented layer-parallel processing approach by comparing it with a *layer-by-layer* (sequential) processing approach using the example of a CNN for the MNIST benchmark.

In the following, an overview of related work is given, where we present previously proposed CNN acceleration and architecture approaches (Section 2). Subsequently, the basics of CNNs are described with a focus on the convolutional layer (Section 3.1). After that, layer fusion is proposed as a method for reducing the storage consumption (Section 3.2), followed by methods to reorder the computations of the CNN (Section 3.3) to achieve a maximally overlapped processing of multiple layers. These methods are particularly useful and also necessary to reduce the storage requirements given by memory-constrained TCPAs and for best exploitation of parallelism (Section 4). Next, we develop a calculus for performance and memory requirements for our proposed mappings (Section 5). Afterwards, the performance and resource requirements of our layer-parallel mapping are applied to a CNN for the introduced MNIST benchmark and compared with a layer-by-layer implementation (Section 6). Finally, we derive the calculus to determine the minimal PE number to satisfy a given throughput (Section 7).

2. Related Work

In general, most CNN accelerators in literature propose sequential layer processing, see, e.g., [5], [8], [9] which incurs significant on/off-chip transfers, sacrificing performance. Layer fusion (Section 3.2) reduces off-chip transfers by exploiting the pipelining potential of the layers of a CNN [10]. For the convolutional layers, loop permutation and unrolling is applied to increase data locality and to utilize the parallelism of the architectures like in [9].

In [9], Zhang *et al.* propose a polyhedral-based dependency analysis, where they classify the data dependencies between different loop dimensions on a given array. However, no layer-parallel processing is considered in this work and the loop transformations differ from ours.

In [10] and [11], layer fusion was proposed for FPGA architectures. FPGAs consist of different reconfigurable units, like logic blocks, DSPs, on-chip interconnections, and memory blocks. Unfortunately, there are several penalties in operation frequency and energy consumption compared to a hardwired application [8]. In addition, we exploit the approaches of layer-parallel processing by reducing the starting times further, such that each subsequent layer can start with computation as early as possible.

For coarse-grained reconfigurable architectures (CGRAs), there exist different compilation-based code generation approaches for the proposed architecture [5], [8]. EMAX [8] proposes an energy-efficient CGRA implementation to accelerate CNNs, with even the ability of training (by switching the instruction set for the training phase). However, no loop transformations or layer-parallel processing is proposed to utilize the pipelining capability of CNNs to the fullest.

In [5], an auto-tuning framework for accelerating CNNs with modulo scheduled CGRAs is proposed. Hereby, it is mostly concentrated on loop transformation, like unrolling and permutation of the loops. However, also here, no layer-parallel processing as in our following approach is considered.

3. Background

3.1. Basic Principle of CNNs

Convolutional Neural Networks (CNNs) are the state-of-the-art machine learning technique for image recognition. A CNN is composed of a set of layers of different types (i.e. pooling, fully-connected, or activation layers), where the most specific layer type is the convolutional layer which characterizes the CNN. An example of a CNN is depicted in Fig. 1(b). In a CNN, the convolutional layers are the computationally most expensive part [9]. In modern CNNs, the convolutional layer is described as a convolution block which consists of different types of convolutions like the standard convolution, the depth-wise convolution, or the pointwise convolution. In this work, we concentrate on the standard convolution, because proposed methods for exploiting parallelism like layer-parallel processing and loop transformations can be equally applied to the other types of convolution blocks [11].

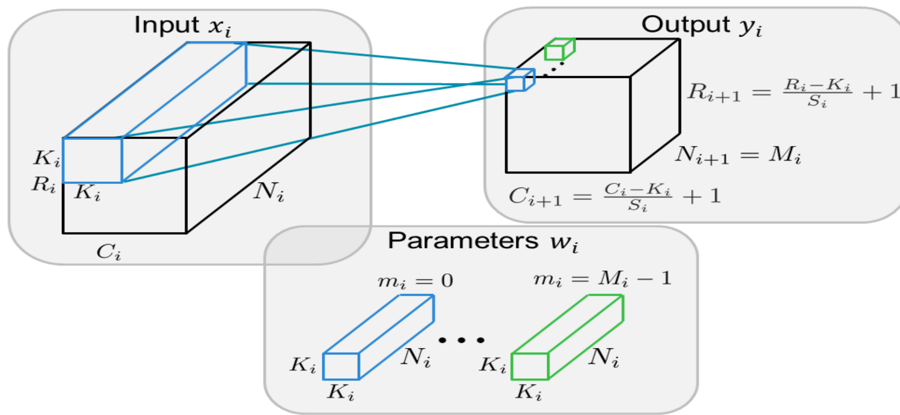


Fig. 2. Scheme of a standard convolution in a CNN. From a set of N_i input feature maps as input (left), a set of M_i output feature maps (right) is computed based on the computation shown in Eq. (1).

In a standard convolution (see Fig. 2), the input $x_i \in \mathbb{R}^{R_i \times C_i \times N_i}$ of a layer i consists of N_i feature maps of spatial dimension $R_i \times C_i$ (rows times columns). Each input feature map $n_i, (0 \leq n_i < N_i)$ is convolved by M_i filters with learned weights $w_i \in \mathbb{R}^{M_i \times K_i \times K_i}$ for a kernel size of K_i (note: for convenience, the kernel size is assumed to have the same size in each dimension. The presented calculations are also valid for different dimensions). For each filter m_i, a bias $b(m_i) \in \mathbb{R}$ is added, which also corresponds to learned weights. This creates M_i output feature maps $y_i \in \mathbb{R}^{R_{i+1} \times C_{i+1} \times N_i}$ which become the input of the successive layer $i+1$ $(y_i = x_{i+1})$.

The following equation describes the computation of one output value in the output feature map $m_i, (0 \leq m_i < M_i)$ in row $r_{i+1}, (0 \leq r_{i+1} < R_{i+1})$, and column $c_{i+1}, (0 \leq c_{i+1} < C_{i+1})$,

$$y(m_i, r_{i+1}, c_{i+1}) = \left(\sum_{n_i=0}^{N_i-1} \sum_{k1_i=0}^{K_i-1} \sum_{k2_i=0}^{K_i-1} w(m_i, n_i, k1_i, k2_i) \cdot x(n_i, S_i \cdot r_{i+1} + k1_i, S_i \cdot c_{i+1} + k2_i) \right) + b(m_i) \quad (1)$$

With stride size S_i, the step width with that the filter is moved over the feature map. For this formula, a pseudo-code is visualized by Algorithm 1. Equation (1) equally describes also pooling layers and the fully-connected layers. In case of a fully-connected layer, the computation in Eq. (1) simplifies to the special case $S_i = K_i = R_i = C_i = 1$.

The pooling layer, which also can be seen as a down-sampling layer, because it always is applied with a stride, $S_i > 1$, has the effect of reducing the spatial dimension (R_{i+1}, C_{i+1}) of the feature map (seen at the

division by S_i in the equations for the output dimensions in Fig. 1(b)). For the pooling layer, the number of filters equals M_i=1.

3.2. Layer Fusion

Layer fusion [10] denotes the overlapped processing of multiple layers of a CNN for a given input. We will exploit this method to drastically save intermediate memory needed between the layers in the proposed TCPA implementations as well as for optimizing throughput.

The first layers of a CNN generally increase the feature map size. For example, in Fig. 1(b), compared to the input feature map (28x28 input image), the first convolutional layer (Conv0) increases the number of feature maps by the number of filters, thus by 24x ($M_0 = 24 = N_1$, see Fig. 2). This increase of the feature maps is common in CNNs, where the trend is to have more layers, with an increasing number of filters. Now, if layers are computed sequentially (layer-by-layer), it is most likely that the output data needs to be stored in off-chip memory. To decrease the amount of off-chip/on-chip transfers, layer fusion may be applied, which was first proposed by Alwani *et al.* in [10].

In our proposed CNN according to Fig. 1(b), after the first convolutional layer (Conv0), the next layer, a pooling layer (Fig. 1(b), Pool1), reduces the spatial dimension (R, C) by a factor of 4x, which is typical for pooling layers (see Section 2.1). Thus, if the pooling layer can start with its computation earlier, we might significantly decrease the amount of data that needs to be stored between the layers.

Dependence analysis reveals that each output value of a convolution (or pooling layer) i depends on a small window of the previous layer $i - 1$. For example (see Fig. 3), suppose that layer i has a window size of $K_i = 3$ (dashed black lines) and a stride of $S_i = 1$, the second $K_{i+1} = 2$ and S_{i+1}=2, respectively. Obviously, one output value in y_{i+1} depends on the input values of x_{i+1} located in the filter window of size K_{i+1} (shown by the 2x2 red square). One value in x_{i+1}, which equals the output y_i of the previous layer, depends on the input values, in x_i located in the filter window, of size K_i (shown by the black dashed square). In summary, one output value in y_{i+1} depends on the input values in x_i, located in a window of size D_i=K_{i+1}+$K_i - 1 = 2 + 3 - 1 = 4$ (seen in the 4x4 red square in the input on the left side). The "-1" is subtracted at the end because we have one overlapping area, when shifting the window by S_i=1. Here, D_i denotes the size of one dimension of a so-called *receptive field*. According to [10], with

$$D_i = D_{i+1} \cdot S_i + K_i - S_i, \tag{2}$$

One can compute recursively the size D_i of the *receptive field*, describing the size of the window in the input layer i, needed to compute one output value in layer $i + l$.

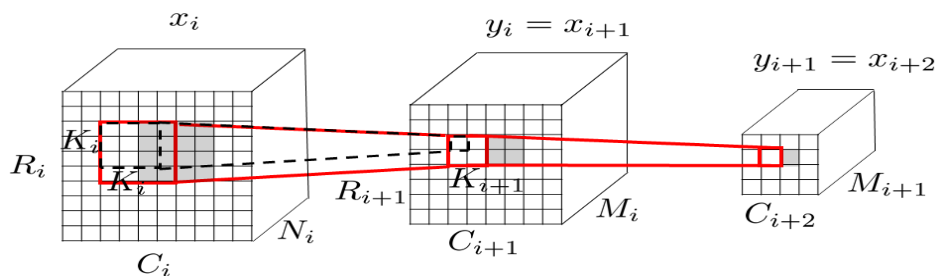
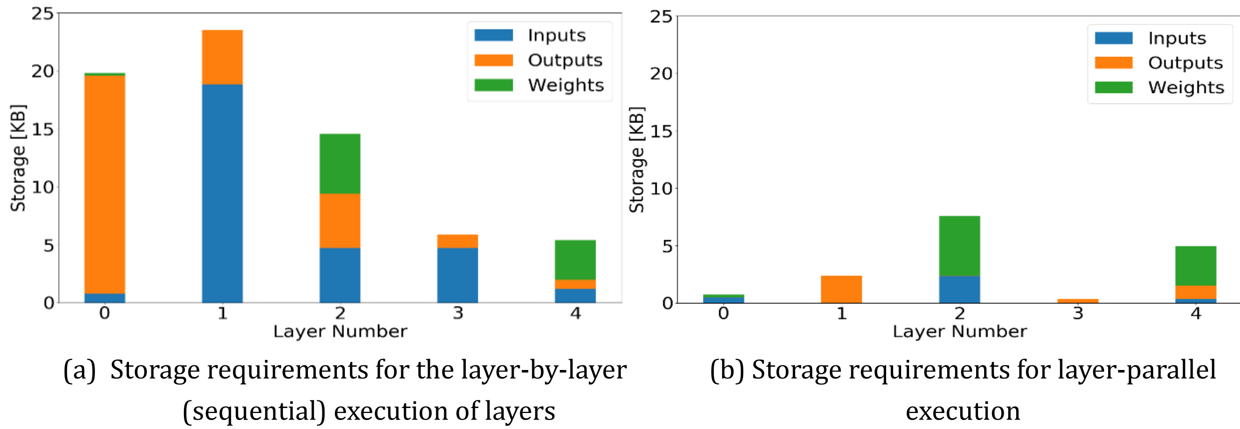


Fig. 3. Example of two layers being fused. One output on the right (in y_{i+1}) depends on a 4x4 *receptive field* of the input on the left (in x_i), which is shown by the red rectangles. The gray area marks the receptive field for the next output value.

With layer fusion, the storage transfers between the TCPA and periphery is decreased (for some nets up to

95%), resulting in a higher computation-to-communication ratio (FLOP/byte access) as well as performance [10].

Fig. 4 provides a preview of the storage requirements for our proposed net (Fig. 1(b)) when processing all layers in parallel (Fig. 4(b)). Because the pooling layers have no weights, and the intermediate data is nearly irrelevant (see Section 5.4), one can observe only tiny storage requirements for the pooling layers (inputs). In the layer-parallel version (Fig. 4(b)), we only need to store small amounts of intermediate results (see Section 5.4 for the calculation), marked as the inputs for each layer. As can be seen, the storage requirements drastically decreased in the layer-parallel version.



(a) Storage requirements for the layer-by-layer (sequential) execution of layers

(b) Storage requirements for layer-parallel execution

Fig. 4. Storage requirements for the convolutional layer and pooling layers of the proposed net, consisting of 3 convolutional layers (0, 2, 4) and two pooling layers (1, 3) in between of them. For each layer, the storage requirement for the cases of (a) sequential layer-by-layer processing, computed using Eq. (14) and (b) layer-parallel processing, computed with Eqs. (16) and (17), is shown.

Another positive effect of layer fusion is that it enables the full pipelining potential (execution of layers in parallel) of a CNN. However, we will show in the next section that, to maximize the overlapping of processing of consecutive layers, the order of computations can also be restructured such that the subsequent layers can start with their computation as early as possible.

3.3. Loop Transformations

In order to allow for a maximal degree of overlapped processing of the CNN layers (if implemented in parallel), the data dependencies between successive layer computations need to be carefully examined. Algorithm 1 shows the pseudocode of a standard sequential loop nest for the computations of one layer according to Eq. (1).

Algorithm 1. Pseudocode for a standard sequential convolution derived from Eq. (1)

```
1  for (m = 0; m < M; m++)       //number of output feature maps (#filters)
2    for (r = 0; r < R; r++)     //number of output feature map rows
3      for (c = 0; c < C; c++)        //number of output feature map columns
4        for (n = 0; n < N; n++)      //number of input feature maps
5          for (k1 = 0; k1 < K; k1++)      //filter kernel dimension 1
6            for (k2 = 0; k2 < K; k2++)    //filter kernel dimension 2
7              output[m][r][c] += weights[m][n][k1][k2] *
                                  input[n][S*r+k1][S*c+k2];
8        output[m][r][c] += bias[m];
```

Obviously, layer $i+1$ can only start once the receptive region of the previous layer has finished being computed. Thus, instead of processing one convolution filter over the whole input image, it would be much better to compute multiple filters in an interspersed way instead. This may be achieved by loop permutation, resulting in the transformed code shown in Algorithm 2. Apart from moving the loop with iterator m to the innermost position, also iterator n has been permuted. Another advantage of this schedule is the potential of reusing input data when processing the same input region for the total of M filters.

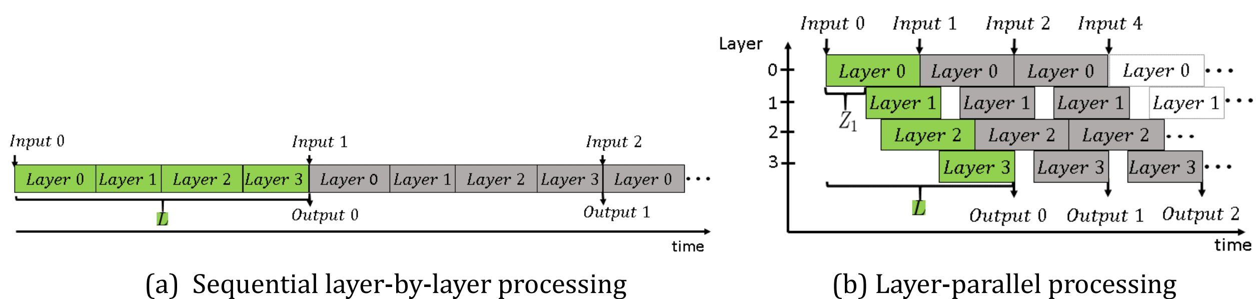
Algorithm 2 describes the transformed loop nest in which loop permutations have been applied to utilize the pipelining potential enabled by layer fusion, and achieving a broadcasting structure for a TCPA architecture implementation. Another notable advantage of this parallelism strategy is that for common deep CNNs, the number of filters, relating to the parameters M, N, are increasing whereas the feature map dimensions (R, C) shrink, such that $M, N \gg R, C$.

Algorithm 2. Pseudocode after loop permutation, by moving the spatial feature map loops (r, c, k) to the outer and the loops which iterate over the different features and filters (n, m) to the innermost positions

```
1   for (r = 0; r < R; r++)          //number of output feature map rows
2     for (c = 0; c < C; c++)        //number of output feature map columns
3       for (k1 = 0; k1 < K; k1++)        //filter kernel dimension 1
4         for (k2 = 0; k2 < K; k2++)     //filter kernel dimension 2
5           for (n = 0; n < N; n++)        //number of input feature maps
6             for (m = 0; m < M; m++)      //number of output feature maps
7               output[m][r][c] += weights[m][n][k1][k2] *
                                   input[n][S*r+k1][S*c+k2]
8             if k1 == K && k2 == K && n == N
9               output[m][r][c] += bias[m];
```

4. TCPA Mapping

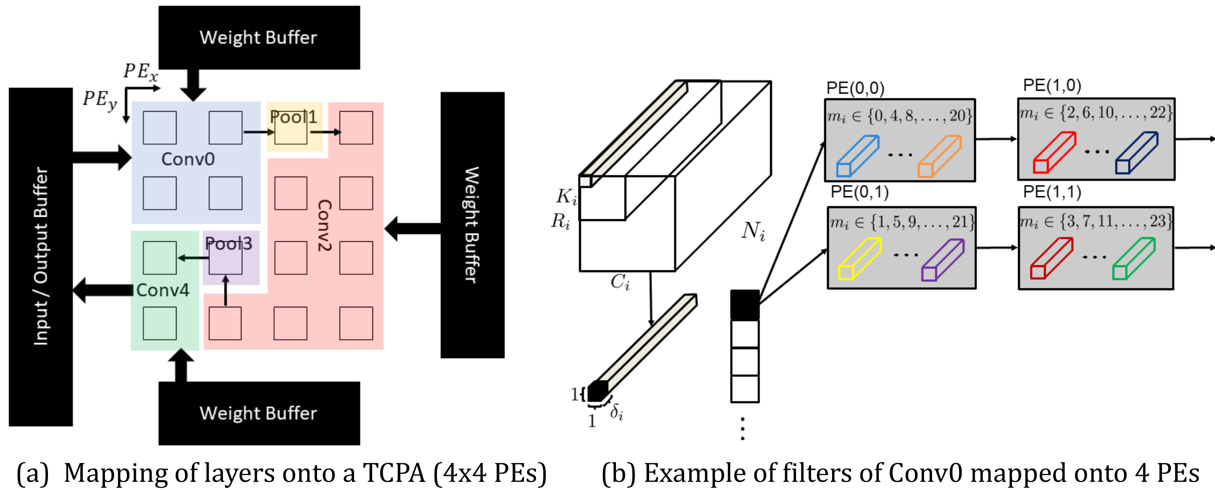
In Section 3.3, we proposed to permute the loop dimensions, such that broadcasting of inputs and the parallel processing of the CNN layers on TCPAs is made possible. The principle of the layer-parallel processing is shown in Fig. 5(b), and is compared against the sequential layer-by-layer processing (Fig. 5(a)). Observe that we can start computing the next input frame before the computations for this frame through all layers are finished (pipelining). Second, the layers are computed in parallel and do not need to wait until the previous layer has finished the computation of all feature maps.



(a) Sequential layer-by-layer processing (b) Layer-parallel processing

Fig. 5. Exploitation of parallelism in processing CNNs. In (a), the conventional layer-by-layer processing is shown. Here, all the layers of the net need to be computed before the next frame (Input 1) can start. In (b), our approach to layer-parallel processing is shown. The approach combines pipelined processing of frames with parallel processing of layers. L depicts the CNN's latency to process one input frame.

Additionally, we may exploit the parallel processing of the layers using different PEs of a TCPA. For that, the inner loops (d, p) are unrolled (Algorithm 3). Moreover, we exploit ILP (instruction-level parallelism) for each PE and loop-level parallelism by using multiple PEs for each layer. Loop-level parallelism is achieved by assigning $\left\lceil \frac{M}{P} \right\rceil$ filters (see Fig. 6(b)) to each PE. Instruction-level parallelism (ILP) is exploited by execution multiple instructions in parallel by parallelizing over the number of input features N, using the δ available functional units. The number of connections, because we need to transfer δ values in parallel between the PEs and the functional units (i.e., number of multipliers and adders for one MAC operation) within each PE determines δ.



(a) Mapping of layers onto a TCPA (4x4 PEs) (b) Example of filters of Conv0 mapped onto 4 PEs

Fig. 6. Example of a layer-parallel mapping of five layers of the CNN in Fig. 1(b) (shown by different colors) to a TCPA (a). The black arrows depict the flow of data. For layer 0, each PE is assigned the computations of $\left\lceil \frac{M_0}{P_0} \right\rceil = \left\lceil \frac{24}{4} \right\rceil = 6$ filters (b). Conv0 is shown mapped to $P_0 = 4$ PEs at location (0,0), (0,1), (1,0), and (1,1) of the TCPA.

Algorithm 3. Resulting loop nest for parallel execution of one layer of a CNN on a TCPA. Shown is the program for PE p, $0 \leq p < P$. Whereas the outer loops $(r, c, k1, k2, n, m)$ are processed sequentially, note that multiple PEs execute multiple filters in parallel. Finally, the innermost loop (d) corresponds to parallel computations performed within each PE (ILP)

```
1    for (r = 0; r < R; r++)         //number of output feature map rows
2      for (c = 0; c < C; c++)     //number of output feature map columns
3        for (k1 = 0; k1 < K; k1++)        //filter kernel dimension 1
4          for (k2 = 0; k2 < K; k2++)    //filter kernel dimension 2
5            for (n = 0; n < N; n+=delta)    //number of input feature maps
6              for (m = p; m < M; m+=P)       //number of output feature maps
7                forall (d = 0; d < delta; d++)
8                  output[m][r][c] += weights[m][n+d][k1][k2] *
                                      input[m][S*r+k1][S*c+k2];
8              if (k1 == K-1 && k2 == K-1 && n == N-1)
9                output[m][r][c] += bias[m];
```

Algorithm 3 depicts the fully parallelized loop program for one layer based on P PEs of the TCPA. Altogether, our approach combines parallelism at three levels: a) Each filter is processed in parallel by

assigning multiple PEs and thus generating one program for each PE. Inside each PE, b) we exploit ILP by using the multiple functional units of each PE. Finally c), different portions of the TCPA execute also the different layers in parallel, see, e.g., Fig. 6(a) (layer-parallel execution). In the following, we present a performance and memory calculus that

1) Determines pipelined implementations to minimize the times Z_i between the start times of subsequent layers.
2) Shows how to reduce the amount of intermediate memory required in layer-parallel processing.

5. Performance and Memory Calculus

In this section, we develop a calculus for both layer-by-layer and layer-parallel execution that formalizes how to calculate the latency L, throughput T, and memory B of a given CNN mapped to a target TCPA. The convolutional and pooling layers are considered to be processed in parallel on the TCPA. Note that fully-connected layers require (in comparison to convolutional layers) few computations but many memory transfers. They thus do not benefit from being accelerated on a TCPA in the described fashion. Therefore, in the following, we assume that fully-connected layers are executed elsewhere (for example, the Processor (LEON3) of a TCPA tile as shown in Fig. 1(a)). We further assume that adequate peripheral hardware is in place to provide all necessary data (weights, intermediate output values) in time.

5.1. Latency of Layer-by-Layer Execution

Since in this mode of execution, each layer is run to completion before its successor starts, the latency L of the CNN, consisting of V layers, is given by the sum of the layer latencies L_i:

$$L = \sum_{i=0}^{V-1} L_i. \tag{3}$$

According to Algorithm 1, each layer i has to compute

$$\underbrace{R_{i+1} \cdot C_{i+1}}_{\text{resolution}} \cdot \underbrace{M_i}_{\text{filter count}} \cdot \underbrace{N_i}_{\text{input depth}} \cdot \underbrace{K_i^2}_{\text{window size}}$$

Operations. However, as described in Section 4, even when executing layer-by-layer, we may still exploit loop-level and instruction-level parallelism by dividing the computations of M_i filters among P_i processing elements and the N_i input feature maps among δ_i functional units, yielding

$$L_i = R_{i+1} \cdot C_{i+1} \cdot \left\lceil \frac{M_i}{P_i} \right\rceil \cdot \left\lceil \frac{N_i}{\delta_i} \right\rceil \cdot K_i^2 \tag{4}$$

Given a clock frequency f of the target TCPA, the throughput T_{sequ} for sequential processing in frames per second is given by

$$T_{sequ} = \frac{f}{L} \tag{5}$$

5.2. Latency of Layer-Parallel Execution

As explained in Section 4 and illustrated in Fig. 5(b), by layer-parallel execution, we denote the pipelining

of subsequent layers of a CNN. Compared to layer-by-layer execution, for layer-parallel execution, we have to consider two peculiarities: 1) layer execution overlaps, meaning that a layer may start as soon as enough output pixels from its predecessor are available, and 2) to never stall this pipeline of layers, all throughputs must be matched. Layer fusion and the loop transformations from Section 3.3 all serve to facilitate this process. Both peculiarities rely on how long it takes a layer i to compute a part of its output such that the next layer can start processing. To calculate a single output pixel, a filter must perform $N_i \cdot K_i^2$ operations, and M_i filters must be computed. In Section 4, we explained how to execute the computation of all M_i filters and N_i input feature maps of one layer in parallel by using P_i processing elements and δ_i functional units. Obviously, it takes

$$z_i^{out} = \left\lceil \frac{M_i}{P_i} \right\rceil \cdot \left\lceil \frac{N_i}{\delta_i} \right\rceil \cdot K_i^2 \tag{6}$$

Cycles to produce the M_i output pixels for each coordinate (r, c). Consequently, the previous layer $i-1$ must be able to provide sufficiently many input pixels to layer i within this time frame to never stall the pipeline. Because both convolving and pooling are window operations, it depends on the stride S_i and kernel size K_i how many new pixels are necessary (the remaining pixels are reused from previous calculations) to compute M_i output pixels once the pipeline is filled. Let:

$$F_i = \min(K_i^2, S_i^2). \tag{7}$$

(There are never more new pixels needed than the entire window.) Consequently, layer $i-1$ takes

$$z_{i-1}^{out} \cdot F_i =: z_i^{in} \tag{8}$$

Cycles to provide the necessary number of $F_i \cdot M_i$ pixels. To not starve the pipeline, it must therefore hold that

$$z_i^{in} \leq z_i^{out}. \tag{9}$$

If this inequality is not satisfied, and as the throughputs must be matched—either the previous layer must be sped up (assigning more PEs, for example), or the current layer must be slowed down (assigning less PEs or throttled by inserting NOPs, for example). Since we assume layer i may start computation as soon as F_i pixels are available, the interval between the start of the previous layer $i-1$ and current layer i is

$$Z_i = z_{i-1}^{out} \cdot F_i = z_i^{in}. \tag{10}$$

The start time t_i and latency L_i of a layer i are then

$$t_i = \sum_{j=0}^{i} Z_j \quad \text{and} \quad L_i = z_i^{out} \cdot R_{i+1} \cdot C_{i+1}, \tag{11}$$

Since a layer i takes z_i^{out} cycles per output pixel, of which it must compute R_{i+1} times C_{i+1}. Accordingly, the Latency L of a CNN, consisting of V layers, is obtained as the start time plus the latency of the last layer $V-1$:

$$L = t_{V-1} + L_{V-1} \tag{12}$$

The throughput T_{par} for the parallel processing of the layers in frames per second is given by

$$T_{par} = \frac{f}{\max_{0 \le i < V}(L_i)} \quad (13)$$

5.3. Memory Estimation

To map a CNN onto a TCPA, its mapping-dependent storage size must not exceed the size of the available on-chip buffers $B^{\max}$. Both layer-by-layer and layer-parallel execution require storing the weights of convolutional layers, which have a size of $B_i^w = M_i \cdot N_i \cdot K_i^2$. Regarding intermediate outputs, for layer-by-layer execution, the entire intermediate output must be stored between each pair of subsequent layers. It follows, the overall required storage for one layer i may be evaluated as:

$$B_i = \max_{0 \le i < V}(B_i^w + \underbrace{N_i \cdot R_i \cdot C_i}_{\text{input size}} + \underbrace{M_i \cdot R_{i+1} \cdot C_{i+1}}_{\text{output size}}), \quad (14)$$

Because entire feature maps $N_i \cdot R_i \cdot C_i$ and $M_i \cdot R_{i+1} \cdot C_{i+1}$ need to be stored, this usually exceeds the available buffer size $B^{\max}$ of the target TCPA and intermediate results must be transferred off-chip.

In contrast, for layer-parallel execution, the weights and intermediate outputs for all layers must fit the on-chip buffers at the same time; however, each layer needs significantly less storage since only part of a feature map must be stored. Furthermore, the first input and last output (the input and output of the CNN) can be stored off-chip without influencing the pipeline. Therefore, it must hold that

$$B = \sum_{i=0}^{V-1} B_i^w + \sum_{i=0}^{V-1} B_i^{inter} \le B^{\max}, \quad (15)$$

where the size of the intermediate values of a convolutional layer i is

$$B_i^{inter} = (D_i - S_i) \cdot C_i \cdot N_i. \quad (16)$$

Since the filter window is slid across the feature map, each input pixel takes part in the computation of multiple output pixels of a filter. We precompute all intermediate products of an input pixel in advance, meaning they must be stored for their entire lifetime. To optimize throughput, the order we compute output pixels corresponds to the receptive field D_i of the layer (see Fig. 3). We advance to the next receptive field, either in horizontal or vertical direction, once the current one is finished. The last precomputed intermediate product is used after $(D_i - S_i)$ rows (or columns, depending on the scanning direction). This holds for all N_i feature maps, yielding Eq. (16).

In pooling layers, on the other hand, no values can be precomputed because they have no weights and usually no overlapping windows. Therefore, pooling layers require an intermediate storage of

$$B_i^{inter} = N_i. \quad (17)$$

The reason is that for each input pixel we can immediately compute the resulting intermediate value (i.e., by adding on previous value) such that we must not wait until the last value for the complete filter window arrives.

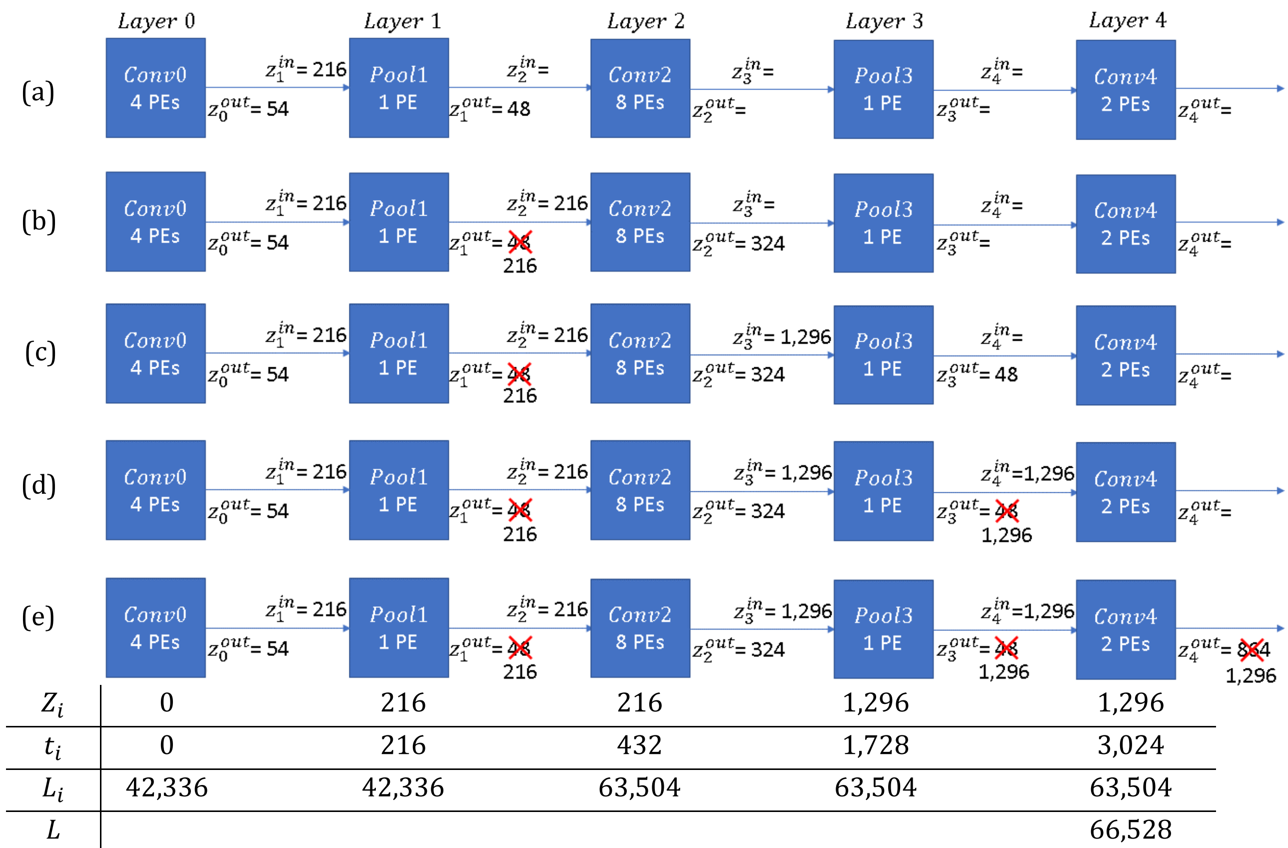
Example:

Assume a word length of 8 bit (1 byte). For the pooling layers with kernel size $K_1 = K_3 = 2$, and stride $S_1 = S_3 = 2$, the windows do not overlap. It is sufficient to store $B_1^{in} = B_3^{in} = N_1 = N_3 = 24$ bytes for each

pooling layer. For convolutional layer 2 (for its parameters see Fig. 1(b)), we need to store $B_2^{in} = \big((D_2 - S_2) \cdot C_2 + D_2 \cdot S_2\big) \cdot N_2 = (8-1) \cdot 14 \cdot 24 = 2.35$ kB, for layer 4 $B_4^{in} = (3-1) \cdot 7 \cdot 24 = 336$ bytes, respectively.

6. Evaluation

In the following evaluation, the proposed net (see Fig. 1(b)) will be mapped onto a 4x4 PE TCPA as shown in Fig. 6(a). The number δ of available functional units within a PE is assumed to be $\delta = 2$. Therefore, two output pixel value computations may be executed in parallel. The configuration time of the TCPA is neglected, because it is only needed to configure the whole TCPA once at the beginning. The first step of mapping the net is to assign a number of PEs P_i to each layer. This is done according to the observation that for a balanced throughput, the PE number of subsequent layers must be chosen to match the different MACs per layer (see Fig. 1(b)). Compared to the convolutional layers, the pooling layers have a low number of MACs, thus 1 PE for each pooling layer is considered sufficient. As we have only 4x4 = 16 PEs and assigning the minimal number of 1 PE for each pooling layer, 14 PEs remain for the three convolutional layers. An example of this distribution is shown in Fig. 6(a). After PE assignment, we now compute the latencies of each layer and the resulting overall throughput. According to Eq.(6), we need $z_0^{out} = \left\lceil \frac{M_0}{P_0} \right\rceil \cdot \left\lceil \frac{N_0}{\delta_0} \right\rceil \cdot K_0^2$ cycles for Conv0 to produce $M_0 = 24$ output pixels. Thus, we obtain $z_0^{out} = \left\lceil \frac{M_0}{P_0} \right\rceil \cdot \left\lceil \frac{N_0}{\delta_0} \right\rceil \cdot K_0^2 = \left\lceil \frac{24}{4} \right\rceil \cdot \left\lceil \frac{1}{2} \right\rceil \cdot 3^2 = 54$ cycles, see Fig. 7(a). In the following, we compute the values z_i^{in} and z_i^{out} for all layers.



	Layer 0	Layer 1	Layer 2	Layer 3	Layer 4
Z_i	0	216	216	1,296	1,296
t_i	0	216	432	1,728	3,024
L_i	42,336	42,336	63,504	63,504	63,504
L					66,528

Fig. 7. Successive adjustments of the latencies regarding Eq. (9). The latencies z_i^{in}, z_i^{out} are given in cycles. In (a) and (b), the latency z_1^{out} of Pool1 needs to be increased to $z_1^{out} = 216$ cycles. In (c), the latencies of Conv2 fulfills Eq. (9) ($z_2^{in} \leq z_2^{out}$). In (d) and (e), the latencies of Pool3, Conv4, respectively, need also to be increased. The table below shows each layer's latency L_i the interval Z_i between the start of the previous layer and the current layer, the starting times t_i of each layer, and the overall latency $L = t_4 + L_4$.

The next layer is pooling layer Pool1 with $K_1 = 2, S_1 = 2$. According to Eq. (7), $F_1 = \min(K_1^2, S_1^2) = \min(2^2, 2^2) = 4$. With Eq. (8), this leads to $z_1^{in} = z_0^{out} \cdot F_1 = 54 \cdot 4 = 216$ cycles needed such that the pooling layer can produce again $N_1 = 24$ output pixels. Here, even with only $P_1 = 1$ PE, the pooling layer PE would be far too fast according to $z_1^{in} \leq z_1^{out}$ as $z_1^{out} = \left\lceil\frac{M_1}{P_1}\right\rceil \cdot \left\lceil\frac{N_1}{\delta_1}\right\rceil \cdot K_1^2 = \left\lceil\frac{1}{1}\right\rceil \cdot \left\lceil\frac{24}{2}\right\rceil \cdot 2^2 = 48$ cycles. This means that z_1^{out} needs to be increased to at least 216 cycles. This may be easily accomplished, e.g., by inserting wait states, NOPs, or other delay. Similarly, we obtain $z_2^{in} = 216$, $z_2^{out} = 324$, $z_3^{in} = 1{,}296 = z_3^{out} = z_4^{in} = z_4^{out}$. For the latency adjustments, regarding Eq. (9), see Fig. 7(b)-(e). Finally, with Eq. (11), $L_4 = z_4^{out} \cdot R_5 \cdot C_5 = 1{,}296 \cdot 7 \cdot 7 = 63{,}504$ cycles and with the start time $t_4 = 3{,}024$ of the last layer, we get the overall latency $L = t_4 + L_4 = 3{,}024 + 63{,}504 = 66{,}528$ cycles according to Eq. (12) until the computation for all layers except of the Fc layer is finished.

With Eq. (13), the throughput for the given CNN, on a 4x4 PE TCPA, is $T_{par} = \frac{f}{\max_{0 \leq i < V} L_i} = \frac{50 \cdot 10^6}{63{,}504} = 787.4$ fps.

For the layer-by-layer case, we give an example for computing the latency for one layer (Conv2). If $P_2 = 8$ PEs are used and $\delta = 2$, with Eq. (4), this results in a latency L_2 of $L_2 = R_3 \cdot C_3 \cdot \left\lceil\frac{M_2}{P_2}\right\rceil \cdot \left\lceil\frac{N_2}{\delta}\right\rceil \cdot K_2^2 = 14 \cdot 14 \cdot \left\lceil\frac{24}{8}\right\rceil \cdot \left\lceil\frac{24}{2}\right\rceil \cdot 3^2 = 63{,}504$ cycles. For the other layers, the latency is computed analogously and with Eq. (3), the overall latency is the sum of the latencies for each layer. Table 1 shows the comparison for the computed latencies for the layer-by-layer execution and the layer-parallel execution.

Table 1. Layer-Parallel vs. Layer-by-Layer. Execution on 4x4 (16 PEs). Layer-by-Layer (1) Corresponds to an Implementation that Uses the Same Number of PEs per Layer as for the Layer-Parallel Case. Layer-by-Layer (2) Uses all 4x4 PEs Per Layer. *Here, on/off-Chip Communication and Configuration Time is Neglected. Layer-Parallel (1) Corresponds to the Layer-Parallel Execution on 4x4 PEs, where Conv2 is the Bottleneck for throughput. In Layer-Parallel (2), four PEs More are Assigned to Conv2, Requiring a TCPA of 4x5 PEs

	Layer-by-layer (1)	Layer-by-layer (2)	Layer-parallel (1)	Layer-parallel (2)
Latency L [cycles]	159,936	55,664*	66,528	44,496
Throughput T [fps]	312.6	898.2*	787.4	1,181.0

In the following, we also show how to compute the minimal number of PEs required for the whole net to satisfy a given throughput.

7. Determining the Minimal Number of PEs for a Given Throughput

Based on Section 5.2, we now determine the minimal number of PEs required for each layer to provide a given throughput T in frames per second at the output. The required throughput in pixels per second for the last layer is therefore

$$T_{pixels} = R_V \cdot C_V \cdot T. \tag{18}$$

With the TCPA's frequency f, the latency z_{V-1}^{out} for computing M_{V-1} output pixels of the last layer must be smaller than $\frac{f}{T_{pixels}}$.

Substituting Eq. (6) for z_{V-1}^{out}, we obtain:

$$\left\lceil\frac{M_{V-1}}{P_{V-1}}\right\rceil \cdot \left\lceil\frac{N_{V-1}}{\delta_{V-1}}\right\rceil \cdot K_{V-1}^2 \leq \frac{f}{T_{pixels}} \tag{19}$$

Hence, we need to determine the smallest value for P_{V-1} satisfying

$$\left\lceil \frac{M_{V-1}}{P_{V-1}} \right\rceil \leq \frac{\frac{f}{T_{pixels}}}{\left\lceil \frac{N_{V-1}}{\delta_{V-1}} \right\rceil \cdot K_{V-1}^2} = \underbrace{\frac{f}{R_V \cdot C_V \cdot T \cdot \left\lceil \frac{N_{V-1}}{\delta_{V-1}} \right\rceil \cdot K_{V-1}^2}}_{W_{V-1}} \tag{20}$$

To calculate the minimal number of PEs necessary for all preceding layers, Eq. (20) can be solved analogously.

Example:

Let us assume we want to obtain a throughput of $T = 100$ fps at the output. Again, assuming a clock frequency of $f = 50$ MHz and $\delta = 2$, we evaluate the minimal number of PEs required to fulfill Eq. (20) starting with layer 4. With $M_4 = 16$, and $W_4 = \frac{f}{R_5 \cdot C_5 \cdot T \cdot \left\lceil \frac{N_4}{\delta} \right\rceil \cdot K_4^2} = \frac{50 \cdot 10^6}{7 \cdot 7 \cdot 100 \cdot \left\lceil \frac{24}{2} \right\rceil \cdot 3^2} = 94.5$ (the output dimensions of layer 4 are $R_5 = C_5 = 7$), the minimal value for P_4 is obtained from $\left\lceil \frac{16}{P_4} \right\rceil \leq 94.5$ as $P_4 = 1$. Analogously, the PE count P_i of the remaining layers is computed. In total, our proposed 100 fps-sustained layer-parallel processing of the net requires only $P = \sum_{i=0}^{V-1} P_i = P_0 + P_1 + P_2 + P_3 + P_4 = 1 + 1 + 2 + 1 + 1 = 6$ PEs.

8. Conclusion

In summary, we have presented an approach for the layer-parallel processing of CNNs on massively parallel processor arrays that allow for the exploitation of different degrees of parallelism in each layer as well as between the layers. High throughput layer-parallel implementations have been derived and compared with corresponding layer-by-layer sequential processing. In the future, we would like to automatically derive PE numbers for each layer as well as to investigate energy efficiency in relation to other proposed parallel architectures.

Acknowledgment

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — Projektnummer 146371743 — TRR 89 “Invasive Computing.”

References

[1] Yann, L., Yoshua, B., & Geoffrey, E. H. (2015). Deep learning. *Nature, 521(7553)*, 436-444.

[2] Shelhamer, E., Long, J., & Darrell, T. (2017). Fully convolutional networks for semantic segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 39(4),* 640-651.

[3] Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). Large-scale video classification with CNNs. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1725-1732).

[4] Norman, P. J. (2017). *In-Datacenter Performance Analysis of a Tensor Processing Unit.* Retrieved from CoRR abs/1704.04760. url: http://arxiv.org/abs/1704.04760

[5] Inpyo, B., Barend, H., Hyemi, M., & Bernhard, E. (2018). Auto-tuning CNNs for coarse-grained reconfigurable array-based accelerators. *IEEE Trans. on CAD of Integrated Circuits and Systems, 37(11),* 2301-2310.

[6] Yu-Hsin, C., Joel, E., & Vivienne, S. (2016). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)* (pp. 367-379). Piscataway, NJ, USA. IEEE Press.

[7] Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., & Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ASPLOS.*

[8] Masakazu, T., Shinya, T. Y., Jun, Y., & Yasuhiko, N. (2015). A CGRA-based approach for accelerating convolutional neural networks. *Proceedings of the IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)* (pp. 73-80). Washington, DC, USA, IEEE Computer Society.

[9] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015). Optimizing FPGA-based accelerator design for deep convolutional neural networks. *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15* (pp. 161–170). New York, NY, USA. ACM.

[10] Manoj, A., Han, C., Michael, F., & Peter, M. (2016). Fused-layer CNN accelerators. *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO)* (pp. 1-12). Piscataway, NJ, USA. IEEE Press.

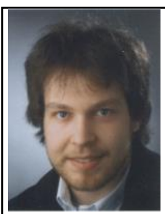
[11] Zhao, R., Ng, H. C., Wayne, L., & Xinyu, N. (2018). Towards efficient convolutional neural network for domain-specific applications on FPGA. *CoRR, abs/1809.03318.*

[12] Frank, H., Vahid, L., Srinivas, B., Alexandru, T., & Oliver, R. (2014). Invasive tightly-coupled processor arrays: A domain-specific architecture/compiler codesign approach. *ACM Transactions on Embedded Computing Systems (TECS), 13(4s),* 1-29.

[13] Éricles, R. Sousa., Alexandru, T., Frank, H., & Jürgen, T. Accuracy and performance analysis of Harris corner computation on tightly-coupled processor arrays. *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP)* (pp. 88–95). IEEE.

[14] Yann, L., & Corinna, C. (2010). *MNIST Handwritten Digit Database.*

Christian Heidorn received his M.Sc. degree in medical engineering from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany, in 2018. Currently, he is working as a scientific researcher at the Chair for Hardware/Software Co-design (FAU). His research focuses on the application of deep learning to co-processors for embedded devices.



Michael Witterauf received his diploma degree in computer science from Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany, in 2014. He is currently employed as a scientific researcher at the Chair for Hardware/Software Co-design (FAU). His research interests include compiler construction and hardware development, especially in the domains of embedded systems and application-specific acceleration.

Frank Hannig received the diploma degree in an interdisciplinary course of study in electrical engineering and computer science from the University of Paderborn, Germany, in 2000; the Ph.D. degree (Dr.-Ing.) and habilitation degree in computer science from Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany, in 2009 and 2018, respectively. He has led the architecture and compiler design group in the Computer Science Department, FAU, since 2004. His primary research interests are the design of massively parallel architectures, ranging from dedicated hardware to multicore architectures, mapping methodologies for domain-specific computing, and architecture/compiler codesign. He has authored or coauthored more than 160 peer-reviewed publications.

Dr. Hannig serves on the program committees of several international conferences (ARC, ASAP, CODES+ISSS, DATE, DASIP, SAC). He is a senior member of the IEEE and an affiliate member of the European network of excellence on High Performance and Embedded Architecture and Compilation (HiPEAC).

Jürgen Teich received the M.Sc. degree (Dipl.-Ing. with honors) from the University of Kaiserslautern, Kaiserslautern, Germany, in 1989 and the Ph.D degree (summa cum laude) from the University of Saarland, Saarbrücken, Germany, in 1993. In 1994, he joined the DSP design group of Prof. E. A. Lee in the Department of Electrical Engineering and Computer Sciences (EECS), University of California at Berkeley, Berkeley, CA, USA (postdoctoral work). From 1995 to 1998, he held a position at the Institute of Computer Engineering and Communications Networks Laboratory (TIK), ETH Zurich, Zurich, Switzerland (habilitation). From 1998 to 2002, he was the full professor in the Electrical Engineering and Information Technology Department, University of Paderborn, Paderborn, Germany. Since 2003, he has been full professor in the Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany, holding a chair in hardware/software co-design.

Prof. Teich is a fellow of the IEEE and member of the Academia Europaea. Since 2010, he is the coordinator of the Transregional Research Center 89 on Invasive Computing funded by the German Research Foundation (DFG).