Enhanced Accelerator Design for Efficient CNN Processing with Improved Row-Stationary Dataflow

Fabian Lesniak
Institute of Information Processing Technology (ITIV),
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
lesniak@kit.edu

Annina Gutermann
Institute of Information Processing Technology (ITIV),
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
gutermann@kit.edu

Tanja Harbaum
Institute of Information Processing Technology (ITIV),
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
harbaum@kit.edu

Jürgen Becker
Institute of Information Processing Technology (ITIV),
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
becker@kit.edu

ABSTRACT

Efficient on-device inference of convolutional neural networks (CNNs) is becoming one of the key challenges for embedded systems, leading to the integration of specialized hardware accelerators in System-on-Chips (SoCs). Due to the memory-bound nature of convolution workloads, it is essential to optimize CNN accelerators for maximum data re-use to reduce memory bandwidth requirements. The row-stationary (RS) dataflow enhances data reuse in CNN processing by storing a subset of input activations, weights and partial sums locally within the Processing Elements (PEs). However, designs of RS accelerators are not publicly available, and many implementation details remain undisclosed. This paper introduces an open-source implementation of a CNN accelerator with RS dataflow. The complete VHDL source code is provided as well as a simulation environment that enables in-depth analysis of different workloads. We contribute an exploration of various design parameters and evaluate their impact on performance. Furthermore, we present an enhanced dataflow that is optimized for parallel processing of convolutions with a high number of channels. Our optimizations yield a performance improvement of up to 2.3x for convolutional layers of common neural networks. An FPGA prototype of the accelerator design, featuring 70 PEs on the Xilinx UltraScale+ ZCU104 platform, achieves 4.012 GOPS at 100 MHz.

KEYWORDS

hardware acceleration, convolutional neural networks, embedded systems, dataflow optimization, row-stationary dataflow

ACM Reference Format:
Fabian Lesniak, Annina Gutermann, Tanja Harbaum, and Jürgen Becker. 2024. Enhanced Accelerator Design for Efficient CNN Processing with Improved Row-Stationary Dataflow. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649476.3658737

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA

ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3658737

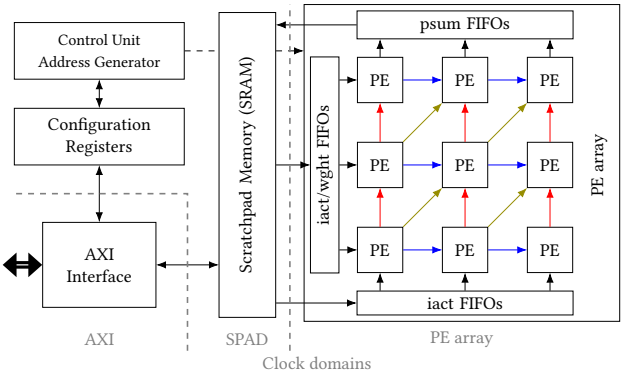

Figure 1: Architecture overview of a 3x3 CNN accelerator implementing a row-stationary dataflow. In the standard configuration, input activations move diagonally, weights to the right and partial sums upwards within the systolic array.

1 INTRODUCTION

One trend has been present for many years already and will certainly persist in the future: the demands on processing power of computing systems are continuously increasing. In the past, growing demands on processors could be met through Dennard Scaling and Moore's Law that guaranteed a steadily increased performance of processors by reducing the transistor size. However, neither principle still applies to the modern development of semiconductor engineering. An additional challenge is the gap between processing power and efficiency in memory access. The so-called memory bottleneck heavily limits applications when faced with increased data demands. A prominent area with high memory and processing demands is the field of deep neural networks (DNNs) that include computationally intensive convolutional layers and frequent off-chip memory accesses. In the embedded field, neither general-purpose processors (GPPs) nor graphics processing units (GPUs) are equipped to handle these requirements due to power limits [1].

A way to enhance performance and energy efficiency for certain tasks is the integration of application-specific accelerators. An example of accelerating data-intensive workloads can be seen in modern smartphones, where on-device inference such as face detection or voice recognition is supported by neural processing units (NPUs) on the device. The performance gains are significant: Snapdragon Mobile Platforms boosted the performance from 3 TOPS for the Snapdragon 845 in 2018 to 26 TOPS for the Snapdragon 888 in 2021 [13]. Other examples of AI accelerators from the industry include the *NVDLA* on the recent NVIDIA Orin System on Chips (SoCs) [10] or external add-on modules like the *Hailo-8* [6].

The need for specialized accelerators is clear. However, the majority of existing AI accelerators are either proprietary with undisclosed implementation details or lack flexibility in their implementation. One of the primary contributions of this work is an accelerator design that utilizes the row-stationary (RS) dataflow used in the closed-source *Eyeriss* accelerator [2]. We provide comprehensive implementation details alongside the full HDL sources and offer a practical approach to implement a high-performance scalable CNN accelerator. Further, we introduce an enhanced RS dataflow optimized for convolution tasks with a high number of channels. Our paper concludes with a detailed evaluation of design parameters and the demonstration of an FPGA prototype.

2 RELATED WORK

Many efforts have been made to develop versatile and performant convolutional neural network (CNN) accelerators, given that convolutional layers account for 90% of the computational load of DNNs [1]. Convolutions in DNNs are a memory-bound problem, making the optimization of data reuse the main target of these accelerators. Most modern CNN accelerators use a systolic array [15] that is built of processing elements (PEs) organized in a 2-D array where data is pushed through as indicated in Figure 1. With a low complexity in control logic and a high degree of both parallelism and data reuse it is well suited for accelerating CNNs as first shown in the *Google Tensor Processing Unit* [9]. The performance of systolic-based accelerators differs greatly depending on the size of the array, the mapping of the workload to the PEs and the dataflow chosen [5, 12]. Depending on the network type and layer size, different choices may be optimal. Therefore, many accelerators offer ways to configure these characteristics.

The before mentioned *NVDLA* offers a wide configuration space including the convolution mode of operation, configurable hardware parameters and the option to include additional SRAM [11]. This tunes the accelerator according to available resources and given workloads. The dataflow is fixed to an adaption of the weight-stationary (WS) implementation and cannot be chosen.

This is different for *GEMMINI* [4], an academic open-source framework used to generate custom hardware accelerators. Here, the dataflow can be set to either output-stationary (OS) dataflow or the WS dataflow. The framework not only generates an accelerator, but provides a full system-level SoC integration that is RISC-V compatible. Tuning the parameters to fit to the given context includes adjusting the array size of the systolic array and defining properties of the scratchpad memory and direct memory access (DMA) unit.

Table 1: Parameters of a convolution problem and loop ranges for parallel and sequential processing.

Variable	Description
H, W	height / width of the input activations
R, S	height / width of the filter kernels
P, Q	height / width of the output matrix
C	number of input channels
M	number of output channels
M0	Number of output channels processed in parallel
C0	Number of channels that fit into the PE line buffers
Q0	Segment of the output width fitting into a line buffer
P0	Segment of the output height fitting the PE array
M1, P1, Q1, C1	Counterparts to the parallel loop ranges which need to be processed sequentially

Eyeriss [2] is a closed-source accelerator originating in academia. It is an energy-efficient reconfigurable accelerator that uses a RS dataflow. The RS dataflow reduces expensive off-chip memory accesses by maximizing data reuse utilizing the PE local storage and inter-PE communication. Eyeriss has been extended as *Eyeriss v2* [3] scaling up the accelerator using a Network-on-Chip (NoC). Unfortunately, most of the internal implementation details are not public and thus findings cannot be used in the open-source ecosystem.

This paper presents a reconfigurable open-source accelerator that is based on a fully parameterizable systolic array. The dataflow can be chosen between the classic *Eyeriss* RS dataflow and a novel adaption of the RS dataflow that scales better for networks with a high number of output channels. We share extensive implementation details and explore various design parameters to establish the accelerator as a basis for future projects in need of a versatile accelerator that implements an energy-efficient RS dataflow.

3 CNN ACCELERATOR DESIGN WITH ROW-STATIONARY DATAFLOW

This chapter describes the design of a CNN accelerator in the form of a systolic array. It is dedicated to perform quantized convolution operations and includes functional units to apply floating-point scaling, offset biasing and activation functions. To maximize data re-use, the CNN accelerator implements a RS dataflow. This reduces memory load caused by reloading input activation data. In addition to the classic RS dataflow, our PE design implements an alternative dataflow that is more efficient for high input channel sizes. Interfaces to other systems include PCI Express (PCIe) and AXI for integration in FPGA-enabled SoCs. While our initial design targets field-programmable gate array (FPGA) platforms, care was taken to avoid FPGA specific IP in favor of an ASIC implementation.

Figure 1 shows an overview of the architecture of the CNN accelerator. It consists of a control block, scratchpad memory and the configurable PE array, shown with only 9 PEs for clarity. The accelerator is designed to operate stand-alone as long as possible in order to reduce communication effort with the host. The control unit implements all convolution loops in hardware so that a

Algorithm 1 Loop nest for the SRS dataflow.

Input: input activations I[H,W,C], filter weights W[R,S,C,M]
Output: output feature map O[P,Q,M]
for m1,p1,q1 in range M1,P1,Q1 **do**
parfor m0 in range M0 ▷ mapped to PE rows
parfor p0 in range P0 ▷ mapped to PE columns
for c1,q0 in range C1,Q0 **do**
parfor r in range R ▷ mapped to PE rows
for s,c0 in range S,C0 **do**
$p = p1 * P0 + p0$
$q = q1 * Q0 + q0$
$c = c1 * C0 + c0$
$m = m1 * M0 + m0$
$w = q + s$
$h = p + r$
$O[p, q, m] += W[r, s, c, m] * I[h, w, c]$
end for
end parfor
end for
end parfor
end parfor
end for

Algorithm 2 Loop nest for the TRS dataflow

Input: input activations I[H,W,C], filter weights W[R,S,C,M]
Output: output feature map O[P,Q,M]
for m1,p1,q1 in range M1,P1,Q1 **do**
parfor m0 in range M0 ▷ mapped to PE rows
for r,c1,q0 in range R,C1,Q0 **do**
parfor p0 in range P0 ▷ mapped to PE columns
for s,c0 in range S,C0 **do**
$p = p1 * P0 + p0$
$q = q1 * Q0 + q0$
$c = c1 * C0 + c0$
$m = m1 * M0 + m0$
$w = q + s$
$h = p + r$
$O[p, q, m] += W[r, s, c, m] * I[h, w, c]$
end for
end parfor
end for
end parfor
end for

complete image with up to 1024 channels can be processed with just one command. PEs are arranged as a systolic array, where input activations are forwarded diagonally from bottom left to top right, weights left-to-right and (partial) sums bottom-to-top. Each PE includes a multiplexer to allow using the vertical datapath for forwarding of input activations as well. The width X and height Y of the PE array can be freely configured to meet workload requirements and technical constraints (available space, resources, etc.). The size of the line buffers within the PEs can also be adjusted. The memory within the PEs is supplemented by the scratchpad memory next to the array, which stores input activations, weights and (partial) sums of convolutions. For most modern CNNs, the scratchpad size does not fit a full convolutional layer and tiling is used to subdivide the problem. The size of the scratchpad memory can be adapted according to the intended use and the available resources, so that a balance of logic and memory requirements is achieved. Configurable clock domain crossing (CDC) is implemented within the PE array buffers and between the AXI interface and the scratchpad memory. This isolates AXI, scratchpad and PE clocks, forming three independent clock domains to achieve maximum performance in each region.

Algorithm 1 describes the loop nest for a classic RS dataflow, which is fully implemented in hardware within the control unit. Refer to table1 for parameter details. This dataflow, which we call the spatial row-stationary (SRS) dataflow, maps columns of the output P spatially onto PE columns and both filter height R and output channels M onto PE rows. Spatially parallelized loops are identified by the keyword **parfor**, while the remaining **for** loops are processed temporally. Each PE buffers partial sums of an output row section W and the corresponding row of filter weights S in a line buffer. Consequentially, one iteration of the loop nest processes a section of the input activation image which fits into the input activation line buffer within the PEs and has to be repeated to process the full input image. Convolution parameters P, Q, C, M are split into inner loop dimensions $P0, Q0, C0, M0$ which fit the line buffers and array size, and $P1, Q1, C1, M1$ in the outer loop which have to be processed sequentially or on another accelerator instance. PE rows are first allocated to process the current filter in parallel (height of the filter R). Afterwards, further filters are mapped to the remaining rows if possible (number of output channels processed in parallel $M0$).

We propose a novel dataflow for RS CNN accelerators, which improves PE utilization in most cases. It especially suits hidden layers of CNNs, which often have a high number of output channels M. Algorithm 2 shows the altered loop nest for comparison.

Compared to the original SRS dataflow, output channels M are mapped spatially to PE rows and output columns P are mapped to PE columns. Filter weights are instead processed in sequence temporally, which requires the PE line buffers to fit RxS instead of only S weights. This overhead is negligible for small filters like $3x3$, especially when existing memory primitives like Block RAM (BRAM) are used. Figure 2 shows a comparison of the SRS and temporal row-stationary (TRS) dataflows. Note that the case shown has been constructed for the sake of clarity, but the principle can be similarly applied to larger dimensions of input activation and different accelerator sizes. Each color denotes a set of weights for one filter that is processed on the respective PE. The numbers in the squares correspond to the input rows being processed in the PE. On the left, four processing steps of the SRS dataflow are shown that process a set of six kernels, which maps to processing three kernels every two iterations. As the PE array size is not a multiple of the kernel height R, the bottom row is not used. Due to input activations travelling diagonally through the array, six PEs do not contribute to the operation as they apply the filter to rows 10, 11, 1 and 11, 2, 2. Overall, 42 % of PEs are idle. Full utilization of the array

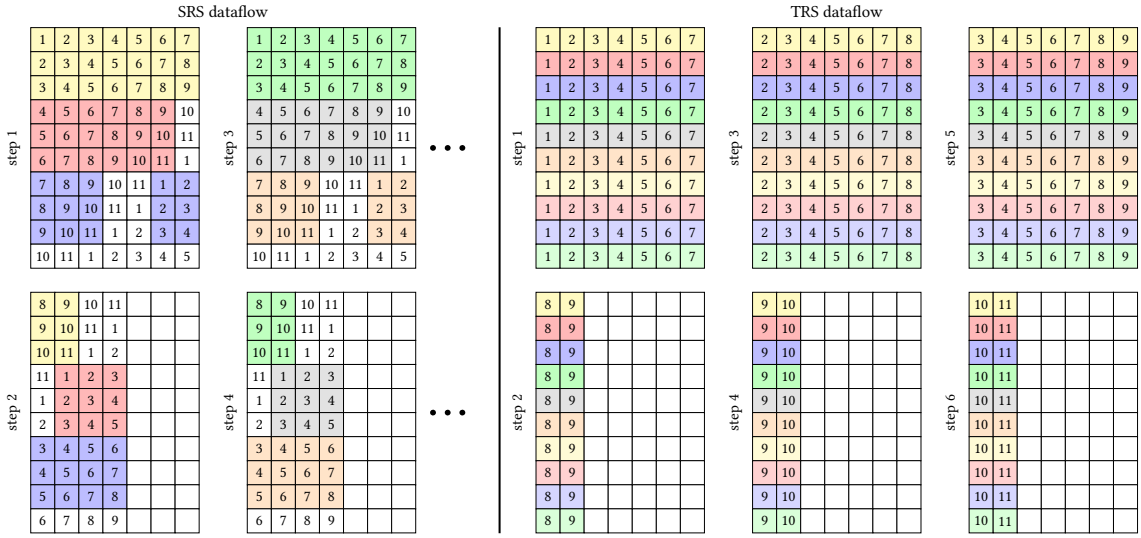

Figure 2: Exemplary mapping of a convolution with input dimensions of $11xW$, kernel size $3xS$ and $C = 10$ on an accelerator with $10x7$ PEs. Each color is one kernel, each square represents one PE and the number inside is the input row being processed in the current step. The left part shows the first four steps in which six kernels are processed using the SRS dataflow, the remaining channels are processed likewise. The right part shows six steps in which all ten kernels are processed using the TRS dataflow.

can only be given when the criteria

$$Y \bmod R = 0$$
$$H \bmod X = 0$$

are met. In comparison, the right side of figure 2 shows six processing steps of the TRS dataflow, in which a set of 10 kernels is being processed. Steps 1, 3 and 5 show full utilization of the array, while steps 2, 4 and 6 have 50 idle PEs each, resulting in 36 % of idle PEs For full utilization, the criteria are:

$$Y \bmod M = 0$$
$$H \bmod X = 0$$

The criteria on Y is easier to fulfill for the TRS dataflow in most cases. The number of kernels processed in parallel M can be set to Y if enough channels are being processed. In the SRS case, however, typical filter heights 1, 3, 5 and 7 have a common multiple of 105, so that very large PE arrays would be required for optimum mapping. Note that for full utilization, the input activation height H shall be a multiple of the accelerator width X. In case of a mismatch, the last step of each iteration has unused PE array columns which is negligible for high input channel counts. The design allows switching between the two dataflow types during runtime, using the vertical data path for input activations for the TRS case. Based on the given criteria, the mapping tool can decide which dataflow is better suited and program the accelerator accordingly.

A driver library on the host provides software interfaces for initialization, configuration and control of the accelerator platform. It allows querying hardware parameters and provides an abstraction layer for common convolution operations. As a part of the library, mapping functions are included to map convolution operations to the accelerator platform, taking the number of available PEs, buffer sizes and input parameters into account.

4 EVALUATION

4.1 Experimental Setup

The CNN accelerator has been fully implemented in VHDL, including AXI and optional PCIe interfaces. Physical implementation results in this section have been obtained using a Xilinx ZCU104 evaluation board with the Zynq UltraScale+ XCZU7EV FPGA. Xilinx Vivado 2023.1 is used for synthesis, implementation and bitstream generation. The prototype design of the accelerator uses a $10x7$ array and 128 kB of scratchpad memory. Driver library and benchmarking software is running on the quad-core Cortex-A53 processor on top of Linux 5.15.19.

Simulation results have been obtained using automated cycle-accurate testing of the RTL model with Siemens Questa Prime version 2022.4. For each simulation run, random input activation and weights are generated and mapped to the accelerator using a greedy scheduling algorithm.

4.2 Evaluation of design parameters

We performed design space exploration for different scenarios to obtain suitable parameters for the PE array size, line buffer size within the PE and input/output FIFO depths. Figure 3 shows the utilization for varying PE array sizes and different convolution problems. The overall utilization per PE is given relative to the total number of cycles for the given parameters for both SRS and TRS dataflows. It can be observed that for small filter sizes of $R = S = 1$, the accelerator performance is limited by memory bandwidth as no input activation data can be reused. The scratchpad clock frequency is fixed at 10 times the PE array clock throughout all simulations and uses one interface to the input activation arbiter. As expected, larger PE arrays, both in X and Y dimension, show degraded performance due to the memory bandwidth bottleneck. A

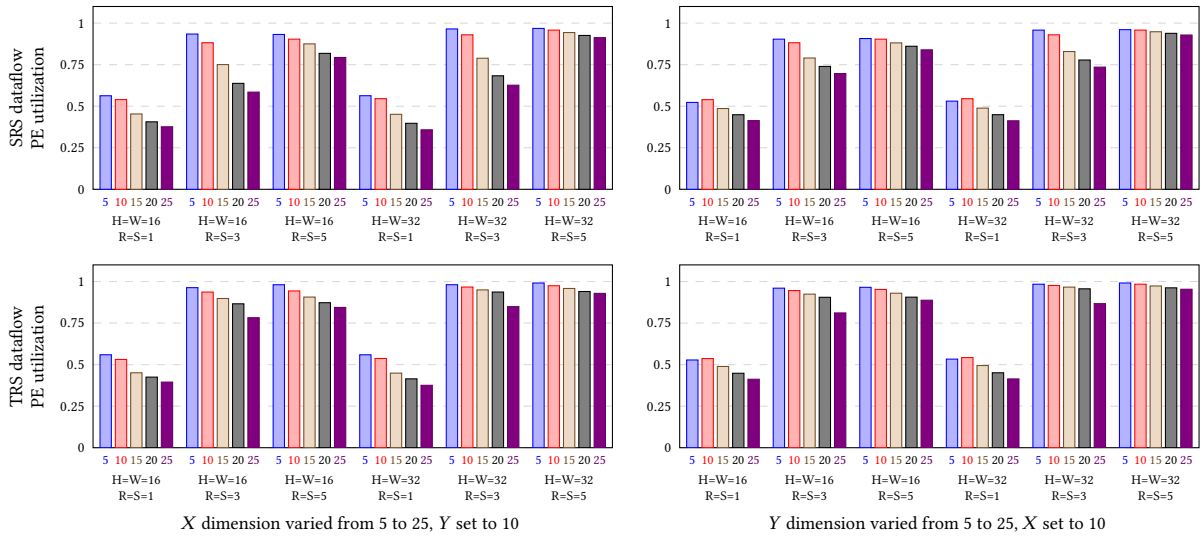

Figure 3: Utilization of the PEs relative to the total number of cycles for varied height and width of the PE array and different convolution workloads. Original SRS dataflow is shown in the upper graphs, our proposed TRS dataflow in the lower graphs.

single input activation arbiter can efficiently distribute data to the number of input FIFOs matching the scratchpad/PE clock frequency ratio. For a high number of input FIFOs, being $X + Y$ for SRS and X for the TRS dataflow respectively, multiple input arbiters should be used for maximum performance if the scratchpad frequency can't be scaled appropriately.

It shows a reduced performance penalty for large array sizes, as only the bottom edge of input activation FIFOs is being used. The impact of limited scratchpad bandwidth is therefore reduced for the TRS dataflow. As expected, larger kernel sizes result in higher utilization of the PEs due to increased data re-use. While $1x1$ convolutions perform similarly for both dataflows, the TRS dataflow can maintain higher utilization for $3x3$ filters.

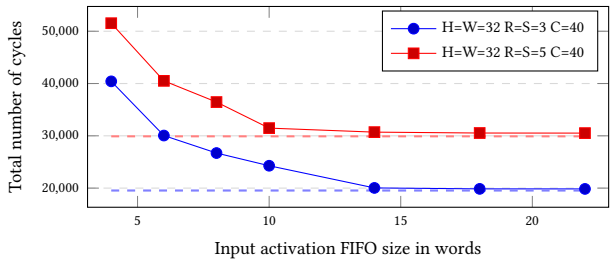

Figure 4: Total number of cycles for processing a 32x32 image, 40 channels, with 3x3 and 5x5 filters on a $10x7$ PE array in TRS mode. The dashed line is the minimum baseline of cycles required for the full workload if no stalling occurs.

The size of the input FIFOs at the western and southern edge is crucial for performance, as the full array needs to be stalled if any FIFO runs empty. Figure 4 shows the result of an input FIFO size parameter sweep from 4 to 22 words on a $10x7$ array. Sizes lower than 10 words result in a great increase in processing time, while more than 16 words hardly reduce the remaining overhead with respect to the theoretical optimum. Output buffers perform best at similar sizes, however their impact on the total processing time is limited due to their brief usage after slice processing.

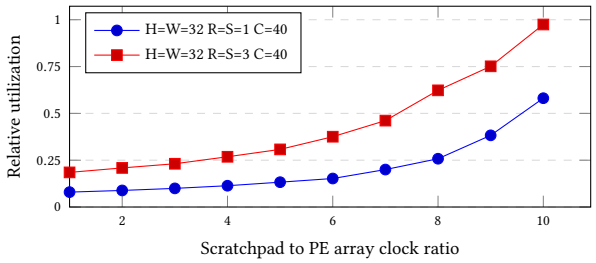

Figure 5: Relative utilization of a $10x7$ PE array in TRS mode for different clock ratios with a single scratchpad arbiter.

The design benefits from a high scratchpad clock, as input activation data is distributed to the array input FIFOs in a round-robin fashion. A single memory port is used to allow using a monolithic SRAM block as scratchpad within each tile. However, the scratchpad could be split into smaller chunks with independent ports if FPGA BRAM is used, reducing the necessary clock frequency through parallization. In any case, the maximum scratchpad interface frequency depends on the FPGA technology. We achieved a scratchpad clock of 250 MHz in our design, with potential for further improvement. This results in a clock ratio of 5:1 by running the PE array at 50 MHz. While this ratio is not ideal yet, further lowering the PE array frequency would still degrade performance.

Carefully balancing the trade-off between scratchpad bandwidth, number of arbiters and input buffer size is essential to maximize performance for the selected array size and clock frequencies. When using a single port scratchpad memory, an array size of $10x7$ is a good compromise with sensible clock ratio, small input buffers are sufficient and common kernels up to $7x7$ do not need tiling.

Table 2: PE utilization for SRS & TRS dataflows for array sizes $10x7$ and $14x12$. Layers from state-of-the-art neural networks are chosen (ResNet-50, GoogLeNet, MobileNetV3).

	H/W	R/S	C	SRS util	TRS util	Workload
$10x7$	28	3	256	82.92%	**90.27%**	R50 2nd last layer
	14	3	1024	76.99%	**85.21%**	R50 2nd last layer
	7	3	512	64.16%	**70.71%**	R50 last layer
	14	1	528	**99.62%**	98.87%	GN 3rd last layer
	7	1	832	**99.05%**	**99.05%**	GN last layer
	28	5	120	**85.71%**	**85.71%**	MN IR layer 7
	14	3	240	77.14%	**85.71%**	MN IR layer 8
	7	5	960	**42.86%**	**42.86%**	MN IR layer 15
$14x12$	28	3	256	61.90%	**69.01%**	R50 2nd last layer
	14	3	1024	42.86%	**98.84%**	R50 2nd last layer
	7	3	512	35.71%	**42.73%**	R50 last layer
	14	1	528	**57.89%**	**57.89%**	GN 3rd last layer
	7	1	832	57.78%	**58.49%**	GN last layer
	28	5	120	47.62%	**92.06%**	MN layer IR 7
	14	3	240	42.86%	**95.24%**	MN layer IR 8
	7	5	960	17.86%	**25.67%**	MN IR layer 15

4.3 Evaluation of SRS & TRS dataflow

We compare our proposed TRS dataflow with the original RS dataflow by Chen et al. [2]. Building a versatile CNN accelerator requires support for different input activation and kernel sizes. ResNet-50 [7], GoogLeNet [14] and MobileNetV3 [8] are chosen as a representative set of CNN models. All these models show the usual procedure for CNNs: the high resolution at the input is reduced quite quickly and fanned out over many channels. Further processing is on with small images and a high number of channels.

Table 2 shows utilization of all PEs for processing of a full higher-dimensional convolutional layer. The results are obtained using the mapping tool for our design, set to use the SRS and TRS dataflow, respectively. As the utilization of a mapping depends heavily on the accelerator shape, we analyzed both our default $10x7$ size and a larger $14x12$ array. It can be seen that the TRS dataflow is at least on par with SRS and exceeds for most inputs.

We measured an end-to-end throughput of 4.012 GOPS with a $10x7$ array at 100 MHz on a workload with $R = S = 3$ and $C = M = 10$ using the TRS dataflow. Taking into account that no workload pipelining was used and time for preload and write-back of data is included, this already comes close to the theoretical maximum of 7GOPS for this configuration.

4.4 Resource usage

Table 3 lists the resource usage of a $10x7$ and a $14x12$ accelerator on a Xilinx Zynq UltraScale+ FPGA, both for the full design and for the individual components. This implementation supports both SRS and TRS dataflows. Removing the multiplexers for dataflow switching saves 6% LUTs per PE. Implementing the full loop nest in hardware reduces host communication and the necessary control and address generation units account for only 4.9% of the resources. The high level of data reuse comes at a cost, so the RS dataflow requires a significant amount of distributed storage: In the $10x7$ design, 52.2% of used BRAM resources are allocated for the PE array. Line buffer size is reduced on $14x12$ to fit the target FPGA.

Table 3: Resource utilization both of the complete design and divided into PE, scratchpad and control unit. Percentages relative to the available resources on Xilinx XCZU7EV.

	Component	LUT	Regs	BRAM
$10x7$	Full design	64.132 *(27.8%)*	36.501 *(7.9%)*	201 *(64.4%)*
	Single PE	849	447	1.5
	Scratchpad	2662	3869	96
	Control Unit	2011	1801	0
$14x12$	Full design	147996 *(64.2%)*	82338 *(17.9%)*	264 *(84.6%)*
	Single PE	849	447	1
	Scratchpad	3342	5349	96
	Control Unit	1993	2352	0

5 CONCLUSION

In this work, we showed an FPGA implementation of a systolic array based convolution hardware accelerator. The RS dataflow is implemented, which enables a particularly high level of data re-use within the array. We introduced the TRS dataflow with significantly improved utilization of the accelerator for most real-world applications. Various parameters of the hardware design were analyzed to highlight the adaptability of the accelerator for different use cases. The HDL design sources are publicly available as part of this publication, making RS accelerators accessible to a broader scientific community. For future work, we aim to create a NoC-based tiled accelerator using this work as the base module and interface High-Bandwidth Memory (HBM) to increase performance.

REFERENCES

[1] Yiran Chen et al. 2020. A Survey of Accelerator Architectures for Deep Neural Networks. *Engineering* 6, 3, 264–274. https://doi.org/10.1016/j.eng.2020.01.007

[2] Yu-Hsin Chen et al. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[3] Yu-Hsin Chen et al. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[4] Hasan Genc et al. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE Press, 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[5] Dennis A. N. Gookyi et al. 2023. Deep Learning Accelerators Configuration Space Exploration Effect on Performance and Resource Utilization: A Gemmini Case Study. *Sensors* 23, 5 (2023). https://doi.org/10.3390/s23052380

[6] Hailo Technologies Ltd. 2023. *Hailo-8™ Century High Performance PCIe Cards.* Retrieved Dec. 14, 2023 from https://hailo.ai/wp-content/uploads/2023/10/Hailo-8-Century-PCIe-Product-Brief-Rev1.1.pdf

[7] Kaiming He et al. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 770–778. https://doi.org/10.1109/CVPR.2016.90

[8] Andrew Howard et al. 2019. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV).* 1314–1324. https://doi.org/10.1109/ICCV.2019.00140

[9] Norman P. Jouppi et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).* 1–12. https://doi.org/10.1145/3079856.3080246

[10] NVIDIA Corporation. 2023. *Maximizing Deep Learning Performance on NVIDIA Jetson Orin.* Retrieved Aug. 16, 2023 from https://developer.nvidia.com/blog/maximizing-deep-learning-performance-on-nvidia-jetson-orin-with-dla/

[11] NVIDIA Corporation. 2024. *Hardware Architectural Specification.* Retrieved Feb. 28, 2024 from http://nvdla.org/hw/v1/hwarch.html
[12] Chan Park et al. 2020. Roofline-Model-Based Design Space Exploration for Dataflow Techniques of CNN Accelerators. *IEEE Access* 8 (2020), 172509–172523. https://doi.org/10.1109/ACCESS.2020.3025550
[13] Hyunbin Park and Shiho Kim. 2023. *Overviewing AI-Dedicated Hardware for On-Device AI in Smartphones.* Springer International Publishing, Cham, 127–150. https://doi.org/10.1007/978-3-031-22170-5_4
[14] Christian Szegedy et al. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 1–9. https://doi.org/10.1109/CVPR.2015.7298594
[15] Rui Xu et al. 2023. A Survey of Design and Optimization for Systolic Array-based DNN Accelerators. *ACM Comput. Surv.* 56, 1, Article 20 (8 2023), 37 pages. https://doi.org/10.1145/3604802

A ONLINE RESOURCES

Repository with HDL sources and simulation files available at: https://github.com/itiv-kit/flexnngine