Data-Rate-Aware High-Speed CNN Inference on FPGAs

Blinded for review

***Abstract*—Dataflow-based CNN accelerators on FPGAs achieve low latency and high throughput by mapping computations of each layer directly to corresponding hardware units. However, layers such as pooling and strided convolutions reduce the data at their output with respect to their input, strongly effecting the data rate of the following layers. This leads to underutilization in fully unrolled designs. While prior work introduced data-rate-aware layer-wise adaptation, determining the most efficient implementation remains challenging.**

This paper presents a data-rate-aware CNN accelerator architecture for multi-pixel processing. Building on existing analytical models, the proposed method performs design-space exploration to identify configurations that improve hardware utilization and resource efficiency while preserving continuous flow of data, keeping all hardware units busy. Experimental results show substantial reductions in arithmetic resources compared to previous designs, enabling efficient implementation of complex CNNs on a single FPGA across a wide range of data rates.

I. Introduction

Convolutional neural networks (CNNs) are widely used in applications requiring low-latency and high-throughput inference, such as autonomous driving [1], speech recognition [2], and high energy physics [3]. Field-programmable gate arrays (FPGAs) are well suited for these workloads due to their ability to exploit fine-grained parallelism and application-specific customization.

FPGA-based accelerators can be broadly divided into three architectural classes. There are fully parallel designs, where the entire input image is processed in parallel [4], [5], [6], [7], [8]. They achieve very low latency, but do not scale to larger networks and cannot implement convolutional layers. Then, dataflow architectures like FINN [9] or hls4ml [10] which process the image sequentially which improves scaling and allow for bigger CNNs to be implemented, but they are not data-rate aware and have a slower target speed. Lastly there are high-speed, data-rate aware architectures like [11], [12], where each layer implementation is tailored to the specific input data rate of that layer. This data-rate-aware layer implementation is important when implementing CNNs as layers such as pooling and strided convolutions reduce the data rate along the processing pipeline, leading to underutilization in fully parallel designs.

Prior work introduced a data-rate-aware, continuous-flow CNN design paradigm [11] that adapts the parallelization and resource sharing of individual layers to the local data rate, enabling high hardware utilization. While this approach achieves a data-rate-matched layer implementation by analyzing the data flow in the model, the approach is not designed to process more than one pixel per clock cycle.

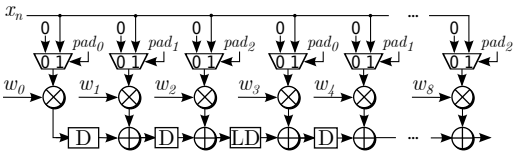


Fig. 1. The KPU base component presented in [11].

This paper extends the continuous-flow paradigm by adapting the approach for multi-pixel processing and simplifies the paradigm by condensing the implementation parameters and constrains.

II. Adapting Continuous-Flow

A. Continuous-Flow architecture

This work builds upon [11], which is described in the following. Each layer ℓ, with $d_{\ell-1}$ input channels and d_ℓ output channels (aka filters or neurons), in the model is implemented in sequence.

Convolutional, and depthwise convolutional layers are implemented using a base component called kernel processing unit (KPU). The KPU takes the kernel weights and computes one sliding window of a feature map per clock cycle. An example of a KPU that processes a 3×3 kernel is shown in Fig. 1 where each input is multiplied with the kernel weights and then summed up with previous weighted features. Each column of multipliers share the same padding select signal pad_i which allows implicit zero padding at the sides of the sliding window. By reconfiguring the weights and adjusting the delays, the KPU can process multiple kernels in sequence.

The pointwise convolutional, and fully connected layers are implemented using a base component called the fully connected unit (FCU) which processes multiple neurons in sequence. An example is shown in Fig. 2 where j features are processed per clock cycle and h neurons are processed sequentially. The input is held for h clock cycles until the next j features are processed, until all $d_{\ell-1}$ features are processed.

Based on the input data rate into the layer $r_{\ell-1}$, which describes how many features have to be processed per clock cycle, a data-rate-matched layer implementation is derived.

For convolutional layers, this is achieved by computing the number of reconfigurations C of the arithmetic-units and the

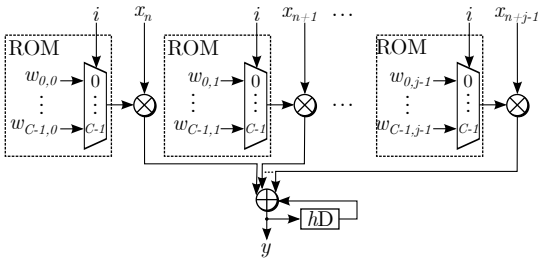
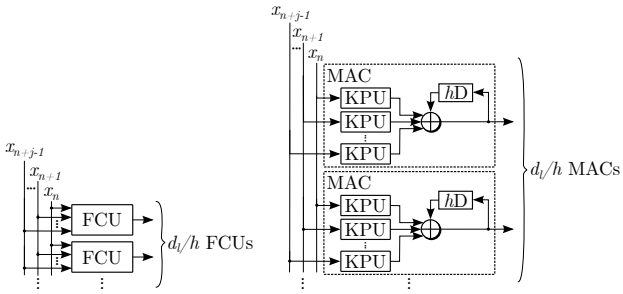


Fig. 2. The FCU base component presented in [11].



(a) Fully connected (b) Convolutional

Fig. 3. The structure of convolutional and fully connected layers.

data interleaving factor I for the current layer. The number of configurations C is directly derived from $r_{\ell-1}$ by

$$C = \min\left(\left\lceil \frac{d_{\ell-1}}{r_{\ell-1}} \right\rceil, d_{\ell-1} d_\ell\right) \tag{1}$$

and the interleaving factor I is derived from C by

$$I = \left\lceil \frac{C}{d_{\ell-1}} \right\rceil . \tag{2}$$

In some scenarios, this leads to rounding errors which results in an architecture that is underutilized.

For fully connected and pointwise convolutional layers, the input data rate $r_{\ell-1}$ is first split up into its numerator j_{max} and denominator h_{max}. The layer implementation is then defined to process j_ℓ inputs over

$$h_\ell = \max(\{h \in \mathbb{N} \,|\, h \text{ divides } d_\ell, h \leq h_{\text{max}}\}) \tag{3}$$

clock cycles.

B. Improved Continuous-Flow Architecture

In the following, we will condense the approach shown in [11] for the data-rate-matched layer implementation starting with fully connected and pointwise convolutional layers and then generalize the approach for convolutional layers and other layer types. Then, we will define the constraints of the architecture parameters, and extend the architecture to process multiple pixels at once.

The Continuous-Flow architecture presented in [11] for the fully connected and pointwise convolutional layers can be condensed to the parameters j and h.

Fig. 3a shows the layer architecture for fully connected and pointwise convolutional layers. Each FCU processes h neurons, so there are d_ℓ/h FCUs in the layer where all FCUs share the same j input signals.

In total, each FCU has to cycle through

$$C = \frac{h \cdot d_{\ell-1}}{j} \tag{4}$$

configurations of weights to calculate each output of its h neurons. So with C configurations, the layer produces

$$C_{r_{\text{out}}} = \frac{h \cdot \frac{d_\ell}{h}}{C} = \frac{d_\ell \cdot j}{d_{\ell-1} \cdot h} \tag{5}$$

outputs per clock cycle and processes

$$C_{r_{\text{in}}} = \frac{j}{h} \tag{6}$$

inputs per clock cycle.

The same approach can be applied on the convolutional layer. Fig. 3b shows how to describe the convolutional layer with j input signals, where each KPU computes h different kernels. The KPUs are grouped together into multiply accumulate (MAC) units, where all outputs are accumulated, and partial results are stored. Each MAC unit consists of j KPUs and is very similar to the FCU itself. By removing the adders in the MAC units, the most common depthwise convolution with a group size equal to $d_{\ell-1}$ can be described too, where the channel multiplier replaces d_ℓ when calculating the number of MACs and the upper limit for h. Other layer types, for example pooling layers, can be described by adapting this depthwise convolution setup further by replacing the KPUs with the base component for pooling layers described in [11].

C. Defining the Constrains of j and h

In [11] the layer parameters for the ℓ-th layer, j_ℓ and h_ℓ, were calculated based on j_{max} and h_{max} where $r_{\ell-1} = \frac{j_{\text{max}}}{h_{\text{max}}}$. We propose to reformulate this problem similar to an upper diophantine approximation of $r_{\ell-1}$ with constrained nominator j_ℓ and denominator h_ℓ. To ensure that the FCUs do not have to process padded or invalid data, the number of input signals into the layer j should be divisible by $d_{\ell-1}$

$$J_\ell = \{j \in \mathbb{N} \,|\, j \text{ divides } d_{\ell-1}\} . \tag{7}$$

Vice versa, the number of neurons processed in each FCU h should be divisible by d_ℓ to ensure that all FCUs process the same number of neurons to eliminate empty times due to synchronization between FCUs, i. e.,

$$H_\ell = \{h \in \mathbb{N} \,|\, h \text{ divides } d_\ell\} . \tag{8}$$

In total, these constrains ensure that if the layer is provided with enough data, the arithmetic units will always process valid data without any empty times wasted for synchronization or additional control circuitry to insert padding data, or filter out invalid data.

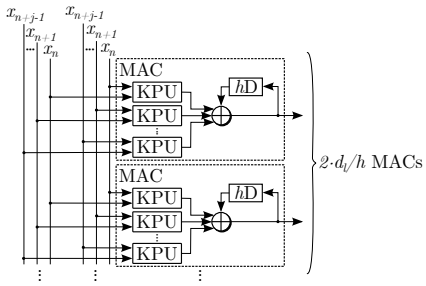


Fig. 4. An example of a convolutional layer implementation that can process two pixels per clock cycle.

D. Defining the layer implementation parameters of layer ℓ

All viable parameter settings for the layer ℓ that satisfy $r_{\ell-1}$ are then described by

$$\mathrm{HJ}_l = \left\{ (j,h) \,|\, j \in J \wedge h \in H \wedge \frac{j}{h} \geq r_{\ell-1} \right\} . \tag{9}$$

The most important criteria for choosing the right implementation setting is how close the implementation gets to the actual input data rate $r_{\ell-1}$ which is described by

$$\mathrm{BestRate}(\mathrm{HJ}_l) = \min \left(\left\{ \frac{j}{h} \,|\, (j,h) \in \mathrm{HJ}_l \right\} \right) . \tag{10}$$

Finally, a parameter setting in HJ_l is selected where $\frac{j}{h}$ is closest to $r_{\ell-1}$

$$(j_\ell, h_\ell) \in \left\{ (j,h) \,|\, (j,h) \in \mathrm{HJ}_l \wedge \frac{j}{h} = \mathrm{BestRate}(\mathrm{HJ}_l) \right\} . \tag{11}$$

In general, it is best to select a parameter setting where h is close do d_ℓ to reduce the number of components (for example KPUs or FCUs) in the layer. This leads to just a few components with big adder trees which can be combined into resource efficient compressor trees [13]. These constrains allow for a better adaptation of the layer to the input data rate rather than in [11] where rounding could occur and the input aggregation is constrained.

E. Adapting the Architecture for Multi-pixel Processing

To allow pointwise convolution and fully connected layer implementations to process two pixels at once, the number of FCUs doubles, where each FCU is processing a specific pixel.

Adapting the convolutional and depthwise convolutional layer to process multiple pixels is more challenging. Fig 4 shows an example of a convolutional layer where two pixels are processed per clock cycle. The term that represents the current input, x_n, has to be adapted to $x_{n,m}$ to address the current pixel index too. Each KPU has to process both pixels to process the current sliding window for the current kernel. Thereby, the KPU has to be adapted to be able to process two pixels at once.

An example of an adapted KPU for a 5×5 feature map with a 3×3 kernel is shown in Fig. 5. The adapted KPU is a non-transposed version of the KPU presented in [11]. So, rather than buffering the weighted partial results, the non-transposed

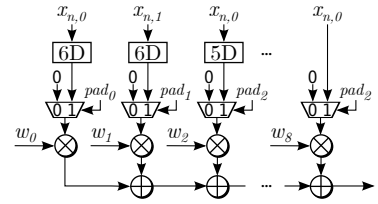


Fig. 5. A non-transposed KPU that can process two pixels per clock cycle.

TABLE I
THE RESOURCES USED TO IMPLEMENT THE MOBILENETV1 MODEL WITH THE SAME DATA RATE.

name	LUT	FF	BRAM	URAM	DSP
[11]	204,931	**563,255**	1,702.5	**0**	5,691
Ours	**158,540**	603,372	**1,449.5**	10	**5,664**

KPU buffers the input features which can be buffered once, and then shared with all other KPUs in the layer. Also, the adders at the bottom shown in Fig. 5 can be reduced into a compressor tree [13]. The input data into the multipliers is delayed to make sure that all multipliers calculate all inputs of the current sliding window at the same time. By analyzing the input image, it can be determined which input signal should be connected to a multiplier and how long the input has to be delayed.

0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0

(a)

0	1	0	1	0
1	0	1	0	1
0	1	0	1	0
1	0	1	0	1
0	1	0	1	0

(b)

Fig. 6. The pixel order of the input image when processing two pixels per clock cycle for a 5×5 input image where the first (a) and second (b) sliding windows indicate how to delay and connect the KPUs to the input signals.

Fig 6 shows the input image where each pixel contains either a zero or one, indicating if the pixel is transmitted over either $x_{n,0}$ or $x_{n,1}$. Considering the time when the last pixel is processed for the first sliding window, which is marked red in Fig. 6a, it can be determined how to delay and connect the two input signals to each multiplier. The last multiplier with weight w_8 is connected to $x_{n,0}$ without any delay. The other multipliers have a delayed input to make sure that all inputs for this sliding window are processed at the time as the last pixel arrives. After calculating the first sliding window, this KPU will then calculate the third sliding window, skipping every second sliding window. Another KPU with a different delay and connectivity pattern is needed to calculate the other sliding windows. The other KPU is thereby derived by the second sliding window shown in Fig. 6b. Now, both KPU designs are repeated for each feature map and for each filter. Or, when performing depthwise convolution with a channel multiplier of one, only once for each feature map.

This approach yields the benefit that, for a stride $s > 1$, some KPU designs may not have to be implemented at all. For

TABLE II
THE SAME MOBILENETV2 CNN MODEL IMPLEMENTED FOR DIFFERENT DATA RATES COMPARED TO THE SOTA.

	F_{max} MHz	FPS Inf/s	Latency ms	LUT	BRAM	URAM	DSP	Power W	P. Eff. mW/Inf.	Precision	FPGA
RTX 3080 GPU**	1860	4105.4	14.93	-	-	-	-	328	79.89	32-bit	
LUTMUL [14]	333	1627	-	529k	1119	-	106	42.12	25.89	4-bit	Alveo U280
[12]	241	4803.1	0.62	136k	1225.5	-	2709	18.04	3.75	8-bit	XCVU37P
FINN [15]*	333	925	-	501k	898	-	106	41.69	45.07	4-bit	Alveo U280
Ours (6/1)	403.71	16,020.40	0.21	186k	1,410	12	6,302	92.34	5.76	8-bit	XCVU37P
Ours (3/1)	404.53	8,026.40	0.42	124k	1,194.5	4	3,168	57.01	7.10	8-bit	XCVU37P
Ours (3/2)	400.64	3,974.61	0.85	77k	1,038	30	1,765	35.62	8.96	8-bit	XCVU37P
Ours (3/4)	405.52	2,011.48	1.66	52k	1,048	19	928	24.87	12.36	8-bit	XCVU37P
Ours (3/8)	408.33	1,012.72	3.30	41k	1,063.5	25	526	19	18.76	8-bit	XCVU37P
Ours (3/16)	410.00	508.44	7.54	33k	1,068	26	306	16.93	33.30	8-bit	XCVU37P
Ours (3/32)	353.48	219.17	14.92	30k	1,078	21	212	14.56	66.43	8-bit	XCVU37P

*FINN implementation results are from **[14]** **Batch size was set to 832, only for the latency the batch size was set to 1

the example shown, with $s = 2$, the second KPU would always produce invalid outputs as the calculated sliding windows are all skipped due to the stride setting and can be removed. A downside of the approach is that the valid output patterns of both KPU designs can be different, leading to additional control circuitry to filter out invalid outputs. If zero padding is used, the padding select signals are also different for each KPU design. But padding select signals and the validity of the KPU outputs can be determined by the position of the sliding window in the feature map, which can be derived by a simple counter in the layer.

III. EXPERIMENTS

All designs from all our conducted experiments were synthesized for an ADM Virtex UltraScale+ (`xcvu37p-fsvh2892-3-e`) as in [11]. We evaluate our approach using MobileNetV1 [16] and MobileNetV2 [17] models, both trained on the ImageNet dataset [18], which enables direct comparison with a large body of existing work. We first implemented the MobileNetV1 with the same data rate as in [11] for comparison.

Table I shows the resource utilization of both approaches. It can be seen that the number of LUTs is reduced tremendously by 22% and BRAM resources by 15%, while raising the number of registers by 7% and slightly reducing DSP resources. These improvements stem from the fact that the proposed approach explores all viable implementations and can therefore select designs that enable larger compressor trees [13] rather than deriving the layer implementation directly from $r_{\ell-1}$.

To further examine the proposed approach, the MobileNetV2 was implemented with different data rates. Table II shows the resource utilization, maximum frequency, power, latency and throughput of each implementation.

It can be seen that by processing two pixels (6 features) per clock cycle, a throughput of almost 16k frames per second (FPS) can be reached. But the approach also allows for much lower rates down to just a few hundred FPS where, in return, resources are saved. The last row shows how an implementation where only 3 features are processed over 32 clock cycles results in only 217 FPS where, in return, only 212 DSPs and 30k LUTs are needed. The table further shows that BRAM utilization is consistently high, independent of the data rate, because all model weights are stored in BRAM. When the data rate is low, fewer weights have to be loaded every clock cycle, which could allow an offloading of those weights to either DRAM or HBM memory.

Finally, we compare our approach to the state-of-the-art (SOTA). Table II also shows SOTA implementations of the same MobileNetV2 CNN model. It can be seen that our multi-pixel accelerator for the MobileNetV2 reaches more than thrice as many FPS as the current SOTA accelerator.

IV. FUTURE WORK

The proposed approach shows promising results for low latency high throughput CNN inference that scales well to moderate sized CNNs. The experiments highlight the flexibility of the proposed approach across different data rates, with lower data rates resulting in reduced FPGA resource utilization. However, as most of the BRAM is used to store the weights of the model, the utilization of BRAM is still very high and does not scale well with the data rate. This can be solved by offloading weights to DRAM for example.

V. CONCLUSION

In this work, an adaptation of a data rate aware CNN hardware accelerator architecture is proposed. We enhance the architecture design space for multi-pixel processing and demonstrate how an implementation of a MobileNetV2 can reach up to 16,020 frames per second. Additionally, we condense the architectural description and reformulate the process of selecting viable implementation parameters, which determine how the layer is reconfigured.

REFERENCES

[1] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi, "Edge computing for autonomous driving: Opportunities and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1697–1716, Aug. 2019.

[2] S. Alharbi, M. Alrazgan, A. Alrashed, T. Alnomasi, R. Almojel, R. Alharbi, S. Alharbi, S. Alturki, F. Alshehri, and M. Almojil, "Automatic speech recognition: Systematic literature review," *IEEE Access*, vol. 9, pp. 131 858–131 876, Sep. 2021.

[3] S. Engel, P. Boutachkov, and R. Singh, "Application of Machine Learning towards Particle Counting and Identification," in *Proc. IBIC'22*, ser. International Beam Instrumentation Conference, no. 11. JACoW Publishing, Geneva, Switzerland, 12 2022, pp. 508–511. [Online]. Available: https://jacow.org/ibic2022/papers/wep42.pdf

[4] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "LogicNets: Co-designed neural networks and circuits for extreme-throughput applications," in *Proc. Int. Conf. Field-Programmable Logic Appl.*, Aug. 2020, pp. 291–297.

[5] M. Andronic and G. A. Constantinides, "PolyLUT: Learning piecewise polynomials for ultra-low latency FPGA LUT-based inference," in *Proc. Int. Field Programmable Tech.*, Dec. 2023, pp. 60–68.

[6] ——, "NeuraLUT: Hiding neural network density in boolean synthesizable functions," in *Proc. Int. Conf. Field-Programmable Logic Appl.*, Sep. 2024.

[7] B. Lou, R. Rademacher, D. Boland, and P. H. W. Leong, "PolyLUT-Add: FPGA-based LUT inference with wide inputs," 2024, Under review.

[8] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking inference in FPGA soft logic," in *IEEE Int. Symp. Field-Programmable Custom Comput. Machines*, Apr. 2019, pp. 26–34.

[9] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, ser. FPGA '17, Feb. 2017, p. 65–74.

[10] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, "hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices," 2021. [Online]. Available: https://arxiv.org/abs/2103.05579

[11] B. for Review, "Continuous-flow data-rate-aware cnn inference on fpga," Preprint, OpenReview, 2026. [Online]. Available: https://openreview.net/forum?id=fQ9QQNwame

[12] Z. Li, Z. Zhang, J. Hu, Q. Meng, X. Shi, J. Luo, H. Wang, Q. Huang, and S. Chang, "A high-performance pixel-level fully pipelined hardware accelerator for neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–14, 2024.

[13] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for fpgas," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.

[14] Y. Xie, Z. Li, D. Diaconu, S. Handagala, M. Leeser, and X. Lin, "Lutmul: Exceed conventional fpga roofline limit by lut-based efficient multiplication for neural network inference," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 713–719.

[15] 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, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3242897

[16] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017.

[17] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019. [Online]. Available: https://arxiv.org/abs/1801.04381

[18] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.