9

TinyHLS, a Python-based Hardware Compiler for 1D and 2D Convolutional Neural Networks

Raphael Gaede[1], Ingo Hoyer[1], Holger Kappert[1], Filippo Milazzo[2], Matteo Cardinali[2], Mauro Roscini[2], Carsten Rolfes[1], and Karsten Seidl[1,3]

[1]Fraunhofer IMS, Germany
[2]AGRICOLUS, Italy
[3]University of Duisburg-Essen, Germany

Abstract

TinyHLS is a Python-based hardware compiler that automatically generates hardware accelerators in the form of hardware description language (HDL) code for Convolutional Neural Networks (CNNs). The description of the CNN architecture as well as the training is done in advance using Python TensorFlow Keras. TinyHLS reduces the development effort to implement inference calculations in digital hardware regarding cost and time. Furthermore, tinyHLS offers a platform independent alternative to commercial high-level synthesis tools like AMD Vivado HLS™ [1]. Each hardware accelerator generated by tinyHLS is a full hardware implementation of its CNN, allowing low-latency and low-power inference. In this work, the concept of this hardware compiler is presented. The workflow of tinyHLS is demonstrated based on a smart farming use case. For this use case a CNN to detect oranges in images is developed in TensorFlow Keras, translated using tinyHLS and implemented on a field programmable gate array (FPGA). The results in terms of accuracy, latency, energy consumption and hardware requirements are then discussed based on the implementation of the use case

CNN. Finally, a brief outlook on the improvement of tinyHLS is given to meet requirements of edge artificial intelligence (AI) computing in the future.

Keywords: HLS, AI, FPGA, Accelerators.

9.1 Introduction

With AI playing a continuously more important role in today's technological world, the demand for resource-limited devices tailored for AI applications increases as well [2]. Hardware-based AI implementations offer greater opportunities for improvement in terms of latency and energy consumption compared to pure software-based implementation approaches [3]. Optimized for individual applications, application specific integrated circuit (ASIC) designs offer high efficiency. However, the development of ASICs is only feasible if the high non-recurring engineering costs (NRE) can be amortized by a large production volume. FPGA-based implementations, however, provide more flexibility at the expense of reduced efficiency [4, 5].

This work introduces tinyHLS, a hardware compiler designed to automatically generate hardware accelerators for CNN models. The primary objective of tinyHLS is the efficient implementation of a given CNN in digital hardware. To achieve this, tinyHLS automatically generates a hardware accelerator in the form of HDL code. A secondary objective is to mitigate NRE by automating the process of hardware accelerator development and verification. Hardware accelerators generated by tinyHLS can be used for both ASIC designs and FPGA programming. Because tinyHLS is platform independent, it provides an alternative to commercial high-level synthesis tools like AMD Vivado HLS$^{\mathrm{TM}}$, that is free from vendor lock-ins.

9.2 Concept

TinyHLS is based on three fundamental conceptual principles. First, a template-based approach is employed to ensure the modularity of the generated designs. Second, intra-layer pipelining is implemented to enhance efficiency. Finally, to optimize resource utilization and energy consumption, tinyHLS incorporates an 8-bit fixed-point quantization.

9.2.1 Template Based Design

As demonstrated by Chao Qian et al. [6], template-based hardware compilers provide substantial advantages. Consequently, this work adopts a

template-based approach. Initially introduced for one-dimensional (1D) CNNs [7], tinyHLS is extended by two-dimensional (2D) convolution, 2D global max pooling (GMP) and dense layer templates in this work. Each layer template is designed as a module in Verilog utilizing the parameters shown in Figure 9.6. The rectified linear unit (ReLU) is employed as the activation function.

The template-based approach introduces a critical design requirement: The input and output signals of each layer template must maintain a consistent structure. This consistency is essential for the convolution and dense layers as it facilitates the sequential connection of multiple instances of these layer types. However, an exception exists for layer types implemented between convolution and dense layers, as these types of layers are instantiated only once.

9.2.2 Intra-Layer-Pipelining

As illustrated in Figure 9.9, the primary computational effort for CNN inference calculations lays within the convolution layers. Consequently, intra-layer pipelining is implemented among the convolution layers and between the final convolution layer and the GMP layer. Intra-Layer-Pipelining is critical for the hardware accelerator's efficiency for several reasons. First, it enables the execution of multiple operations in parallel, leading to faster calculations, albeit at the expense of larger designs. Second, intermediate results from each layer are directly processed by the subsequent layer, eliminating the need to store complete layer results in registers. Third, intra-layer pipelining allows the inference calculation to start during the data transmission process. This saves time and resources as there is no need to save the entire input signal within the hardware accelerator.

The intra-layer-pipelining also introduces an essential design necessity: The templates for convolution and GMP layer must process their input signals within the same time frame. This synchronization ensures that, after a predetermined number of clock cycles, each layer instance receives its input from the preceding layer instance while passing its output on to the subsequent layer instance at the same time. To pass input signals to the first convolution instance, tinyHLS provides an advanced eXtensible interface4-Lite (AXI4-Lite) [8] to transmit data in a synchronized manner.

9.2.3 Quantization

To implement the inference calculation efficiently in digital hardware, quantization is essential. Compared to a 32-bit multiplication, an 8-bit

multiplication for example consumes approximately 15 times less energy and requires roughly 12 times less area [9]. With more aggressive quantization however, the classification performance of the CNN decreases [10]. Therefore, a trade-off between efficiency and performance needs to be established. Given that 8-bit fixed-point quantization is widely employed in embedded AI applications [9, 11] and 8-bit values align well with 32- and 64-bit processors and memories, tinyHLS adopts an 8-bit fixed-point representation for all values, including the weights and bias of the CNN as well as all intermediate results computed by the hardware accelerator.

Images are typically represented with 8 bits per pixel. However, tinyHLS incorporates data preprocessing to quantize input images from 8 bits per pixel down to 4 bits per pixel. This quantization reduces the number of gradations of each colour channel from 256 to 16, while preserving the essential information contained in the image. Utilizing 4 bits per pixel instead of 8 bits reduces the effort required for data transmission and processing. More importantly, with 4 bits per pixel, intermediate results are less likely to overflow, as all intermediate results are constrained to 8 bits. In case of overflow, intermediate results are saturated. To further decrease the computational effort, input images are sampled down to a resolution of 28x28 pixels with 3 colour channels during preprocessing. The effect of the preprocessing is illustrated in Figure 9.2.

The distribution of the weights and bias is analysed for each CNN to determine the most suitable split of 8 bits into integer and fractional bits (Figure 9.4).

To identify the optimal split between integer and fractional bits for all intermediate results, tinyHLS evaluates the CNN performance concerning various 8-bit-fixpoint representations (Figure 9.5). It is important to note that the quantization of the input image, which serves as the input to the first layer, is always configured with 4 integer bits and no fractional bits. In contrast, the output of all layers - and consequently the inputs to all subsequent layers - utilizes 8 bits with a supposedly different distribution of integer and fractional bits. This distinction must be taken into account when initializing the individual layer templates.

9.2.4 Software Model

The evaluation of the CNN performance concerning different 8-bit-fixpoint representations is not executed in hardware simulation. Instead, the hardware accelerator's functionality is modelled in software. Although in software all

operations are performed sequentially, the quantization applied is consistent with that of the hardware accelerator. The key advantage of the software model is its flexibility, allowing for the adjustment of the number of bits allocated for intermediate results, as well as their distribution in integer and fractional bits. This makes the software model a powerful tool for the evaluation of the CNN performance with respect to quantization. Additionally, the software model is used for verification purposes, as described in 1.2.6.

9.2.5 Translation

During the translation process, all values for weights and bias are quantized to 8-bit fixed-point representation and stored in Verilog header files. These Verilog header files are provided to the top module generated by tinyHLS. With the most suitable split into integer and fractional bits for the intermediate results determined prior to translation, the layer templates are instantiated and interconnected in the top module. When instantiating the layer templates, all available parameters according to the CNN architecture must be considered (Figure 9.6).

9.2.6 Verification

For verification purposes, tinyHLS also generates a testbench. That testbench reads an image and passes it on to the top module. Additionally, the outputs from each layer instance are collected within the testbench, allowing for the complete layer results to be stored at the end of the simulation. After the simulation has completed the layer results are written to designated files and imported into Python for verification. Since TensorFlow Keras employs 32-bit floating-point representation for intermediate results [12], while the hardware accelerator utilizes 8-bit fixed-point representation, deviations between the results are anticipated. Consequently, direct comparison of the results from TensorFlow Keras and those from the hardware accelerator is not feasible. Instead, the software model serves as an intermediate verification step, which aims to trace back any discrepancies between the TensorFlow Keras results and those obtained from the hardware accelerator solely to quantization effects. In the first step of the verification process, the results from the hardware accelerator are compared to those from the software model, which is configured for 8-bit quantization with the same distribution of integer and fractional bits as the hardware accelerator. In the second step, the results from TensorFlow Keras are compared to the software model,

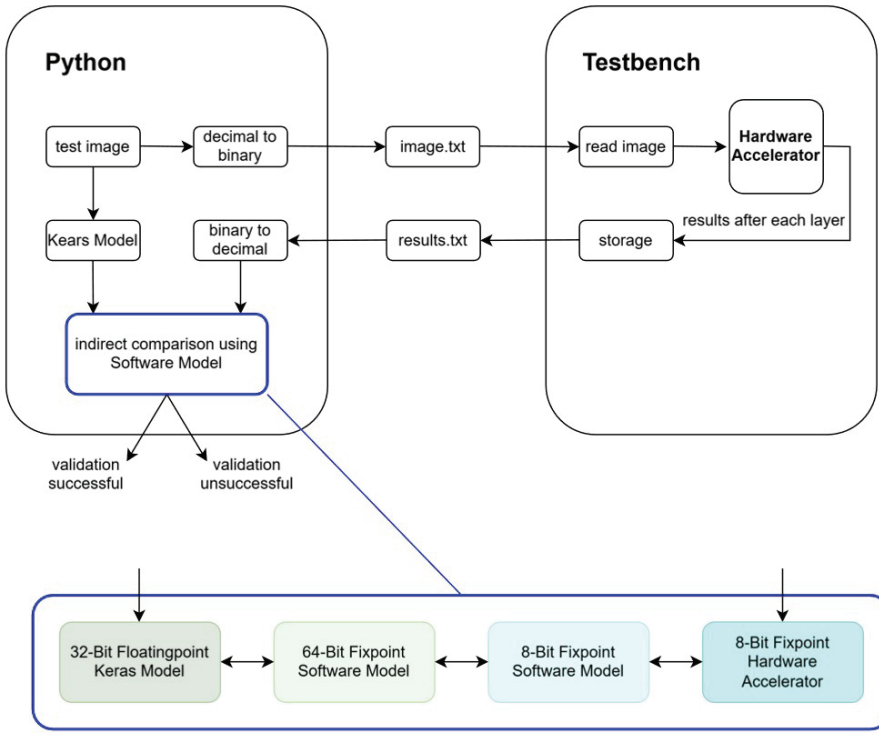


Figure 9.1 Verification Process.

which is set to a total quantization of 64 bits, comprising 32 integer and 32 fractional bits. Each partial verification step is deemed successful only if all results after each layer match. The verification process is illustrated in Figure 9.1.

9.3 TinyHLS Workflow

The tinyHLS workflow is demonstrated based on a use case CNN to detect citrus fruit in images. In the context of smart farming, image classification can facilitate the detection of pests or diseases, allowing for the estimation of crop quality and quantity. In many cases only images containing fruit are relevant to these assessments. To pre-select all images containing fruit, a simple CNN can be trained using TensorFlow Keras, translated using tinyHLS and ultimately executed on an edge device.

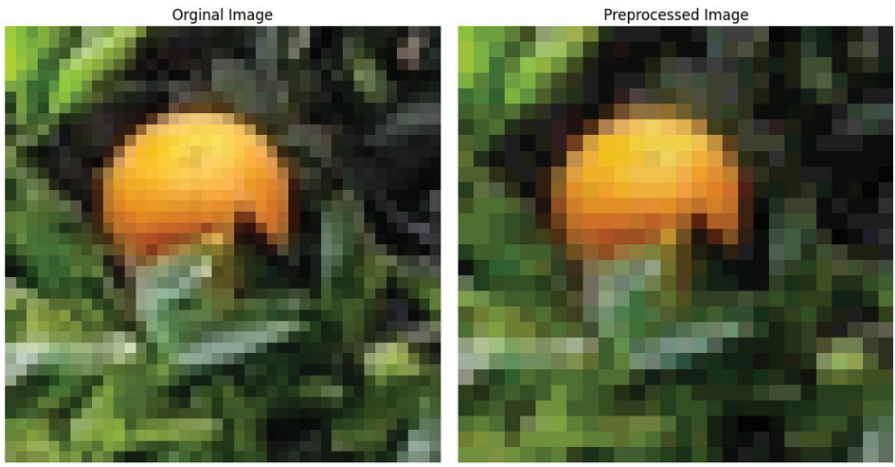


Figure 9.2 Preprocessing of Use Case Dataset.

9.3.1 Preprocessing

For this use case, a dataset comprising 310 images containing at least one orange and 1,220 images without any oranges is utilized. Before training, all images undergo preprocessing which includes rescaling to 28x28 pixels and quantizing each pixel from 8 bit to 4 bit. As shown in Figure 9.2, preprocessing does not discard the information of the image significantly.

9.3.2 Training

With 70 % of the preprocessed dataset serving as training data and the remaining 30 % as test data, the use case CNN, utilizing the architecture described in Table 9.1, is trained over 50 epochs. As illustrated in Figure 9.3, this small CNN architecture, which employs only 864 parameters, is sufficient for this image classification task.

9.3.3 Evaluation

Before translating the trained CNN into HDL code, its performance is evaluated concerning 8-bit quantization. First, the distribution of the weights and bias is analysed to determine the most suitable split into integer and fractional bits. Given that most of the values range from -1 to 1 (Figure 9.4), the 8-bits are divided into 1 integer and 7 fractional bits.

Table 9.1 Use Case CNN Architecture

Layer	***Type***	***Parameter***	***UC CNN***
1	Conv	In. Shape No. Kernels Kernel Shape	[28,28,3] 4 [5,5]
2	Conv	In. Shape No. Kernels Kernel Shape	[24,24,4] 6 [3,3]
3	GMP	In. Shape Out. Shape	[22,22,6] 6
4	Dense	In. Neurons Out. Neurons	6 18
5	Dense	In. Neurons Out. Neurons	18 10
6	Dense	In. Neurons Out. Neurons	10 2

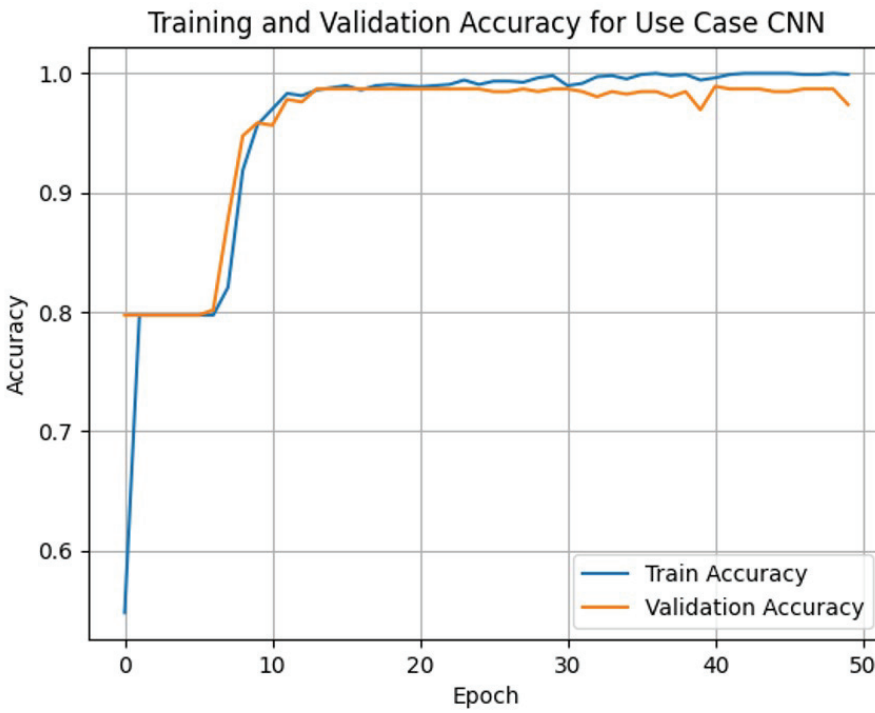


Figure 9.3 Training of Use Case CNN.

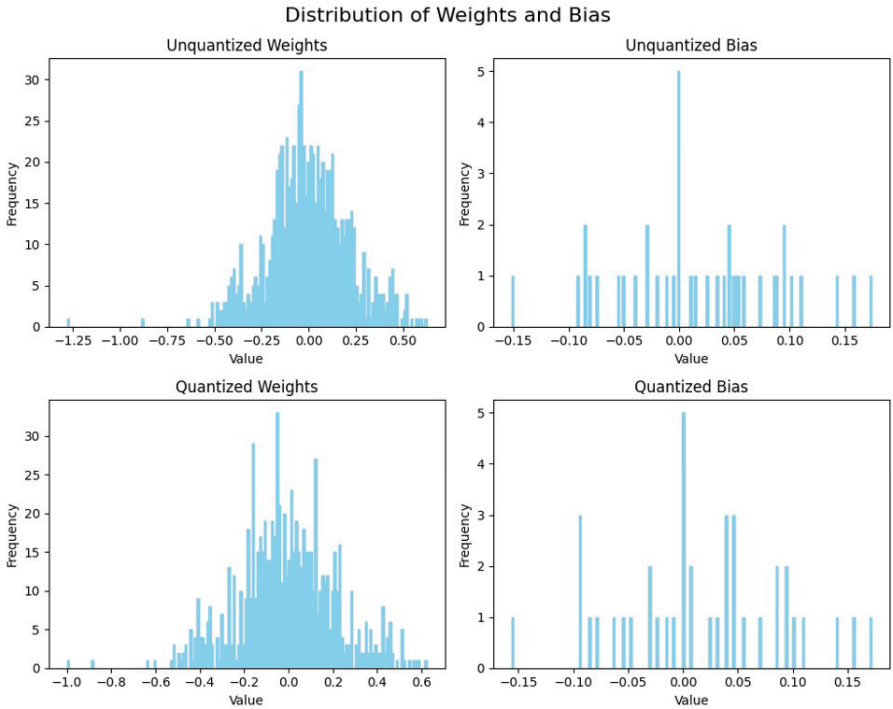


Figure 9.4 Distribution of Weights and Bias of Use Case CNN.

To determine the most suitable split into integer and fractional bits for all intermediate results, the software model is employed to calculate the inference for all 459 test images using various 8-bit fixed-point representations. As illustrated in Figure 9.5, a split into 5 integer and 3 fractional bits is found to be optimal. With an accuracy of 97.59 % the loss in performance due to quantization amounts to only 1.75 percentage points compared to the performance achieved in TensorFlow Keras.

9.3.4 Translation

With the most suitable split for intermediate results determined to 5 integer and 3 fractional bits during the evaluation step of the tinyHLS workflow, the layer templates are instantiated and interconnected in the top module during the translation process of the tinyHLS workflow. These layer templates are instantiated with all respective parameters taken into account. (Figure 9.6). The values for all weights and bias are provided to the top module via Verilog header files.

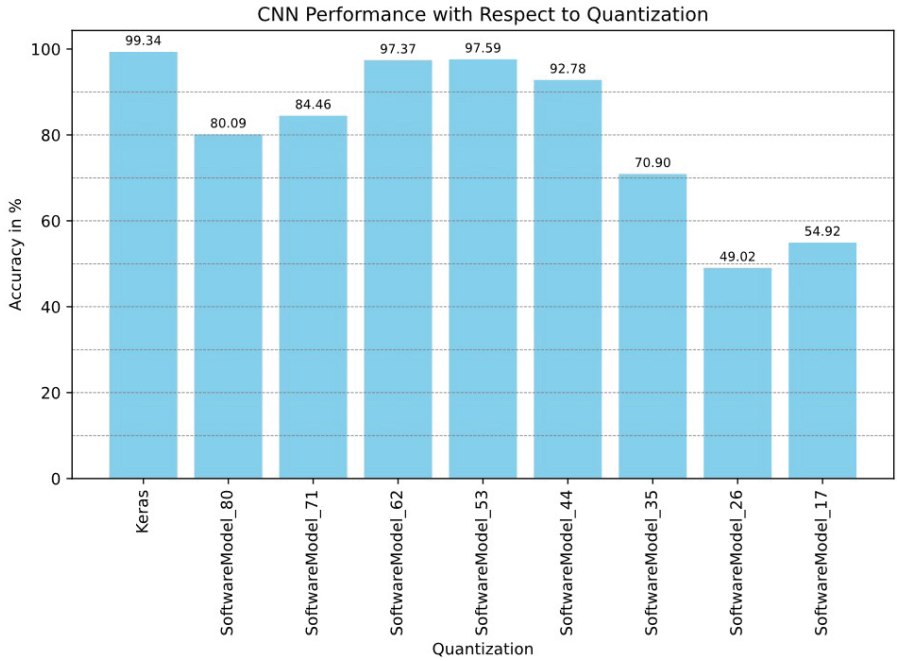


Figure 9.5 Evaluation of Performance of Use Case CNN with Respect to 8-Bit Fixed-Point Quantization.

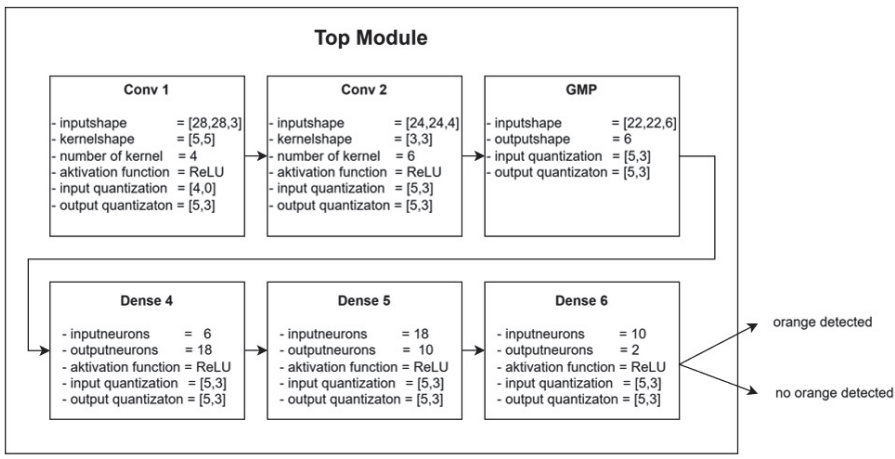
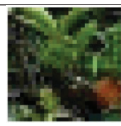
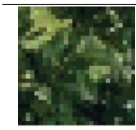
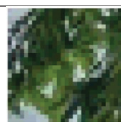


Figure 9.6 Top Module for Use Case CNN.

Table 9.2 Verification Results for Use Case CNN

Image	*Inference Keras*	*Inference SW 64*	*Inference SW 8*	*Inference HWA*	*Verification Successful*
	1.000000	1.000000	0.999981	0.999981	True
	0.000158	0.000158	0.017986	0.017986	True
	1.000000	1.000000	0.999997	0.999997	True
	0.000009	0.000009	0.014064	0.014064	True
	0.000016	0.000016	0.009708	0.009708	True

9.3.5 Validation

The generated hardware accelerator is verified based on 5 test images as described in 1.2.6. As shown in Table 9.2, all 5 test images are verified successfully. Although the inference results of the hardware accelerator are not as confident as the results obtained from TensorFlow Keras, the hardware accelerator classifies all 5 test images correctly.

9.4 Implementation

Due to platform independency, the hardware accelerator generated by tinyHLS can be used for ASIC designs as well as vendor independent FPGA programming. In this work, the hardware accelerator for the introduced use case CNN is implemented on the Kria KR260 Robotics Starter Kit [13] using AMD Vivado$^{\text{TM}}$. As illustrated in Figure 9.7, the Kria FPGA Board comprises a processing system (PS) containing a quad-core Arm Cortex-A53 processor and a programmable logic (PL), where custom logic is located. To

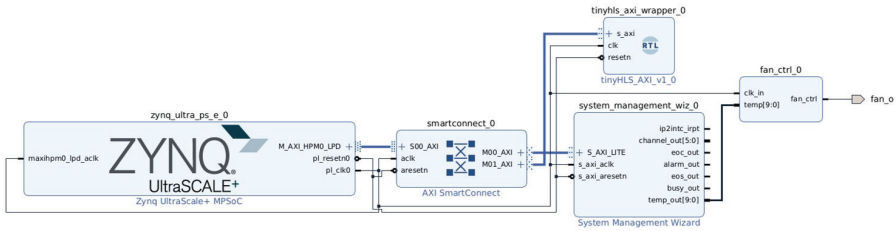


Figure 9.7 Setup on Kria Board.

connect master and slaves via AXI4-Lite bus, the smartconnect intellectual property (IP) block is included in the PL of the board. Besides the processor, which serves as a master, the hardware accelerator and a fan controller are connected to the AXI4-Lite bus as slaves. In this work, both, the fan controller and the hardware accelerator run on a 100 MHz frequency, even though the hardware accelerator can be clocked as fast as 180 MHz.

9.5 Results

The efficiency of the hardware accelerator in terms of hardware requirements as well as latency and energy consumption per inference is assessed based on the implementation of the use case CNN on the Kria board. The results are discussed in the following sections.

9.5.1 Hardware Requirements

Although the CNN utilizes only 864 parameters, its hardware accelerator requires 65,805 look up tables (LUTs) utilizing 56.19 % of the available LUTs on the PL of the Kria board and 27,380 Flip Flops (FFs) utilizing 11.69 % of the available FFs on the PL of the Kria board (Figure 9.8). This large design is a result of extensive parallel computing. Despite the Kria board comprising 1248 digital signal processing (DSP) tiles and 144 block random access memory (BRAM) tiles, none of these specialized resources are utilized in the design. Therefore, the primary reasons for the large design footprint are parallelization and the lack of utilization of specialized tiles.

9.5.2 Latency

For latency considerations, the hardware accelerator is first analysed as a stand-alone implementation meaning that the input image is provided directly to the hardware accelerator without accounting for data transmission via

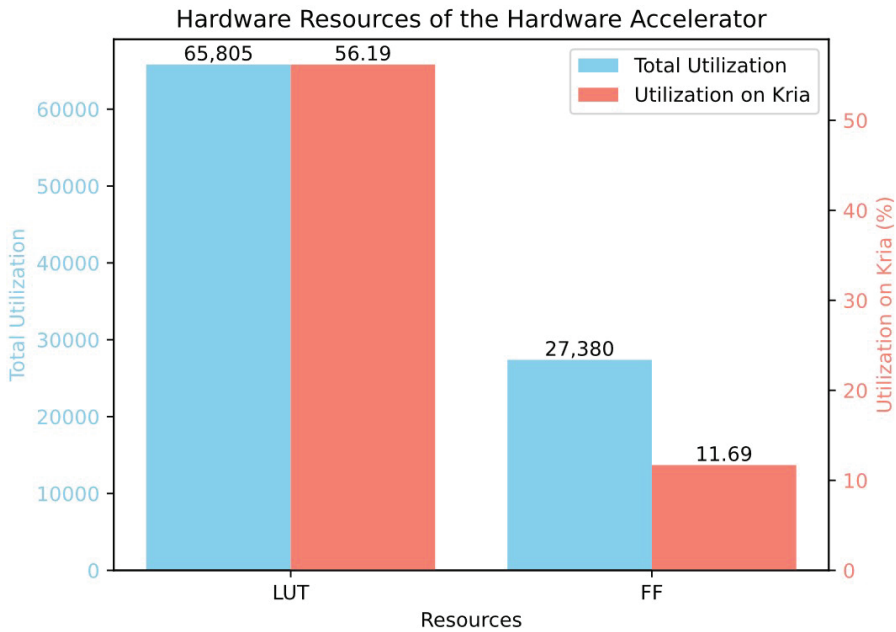


Figure 9.8 Resource Utilization.

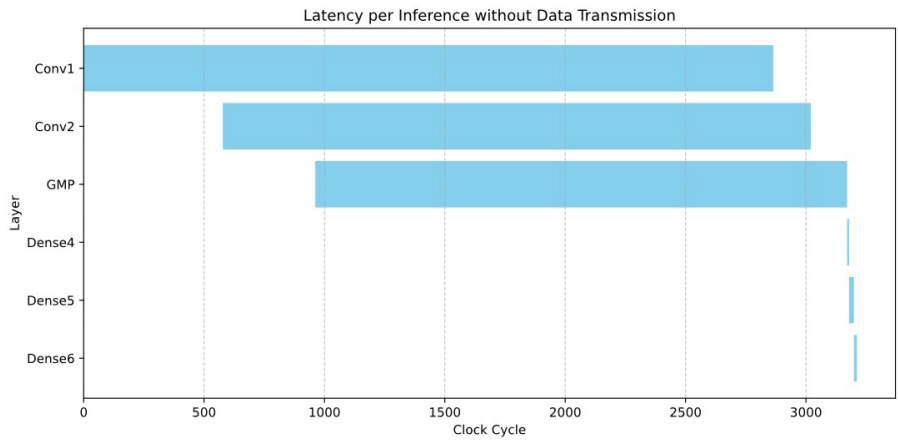


Figure 9.9 Latency per Inference Without Data Transmission.

AXI4-Lite. As shown in Figure 9.9, intra-layer pipelining has a significant impact on latency leading to a total number of 3211 clock cycles per inference calculation.

For benchmarking purposes, the use case CNN is implemented in C code using keras2C [14]. The C code is compiled using GCC compiler with -O0 flag for no optimization and -O3 flag for aggressive optimization. After compilation, the C code is executed on the quad-core Arm Cortex-A53 processor in the PS of the Kria board. As shown in Figure 9.10 the pure software-based implementation on the quad-core Arm Cortex-A53 processor requires 48.563 ms per inference using -O0 flag and 48.538 ms per inference using -O3 flag. With 0.063 ms the hardware accelerator located in the PL of the Kria board requires only 0.13 % of the latency achieved in software.

As the hardware accelerator on the Kria board is clocked at 100 MHz, the latency is expected to be 0.03211 ms considering that 3211 clock cycles are required per inference as shown in Figure 9.9. The longer latency is primarily attributed to data transmission via AXI4-Lite as data transmission is slower than data processing by the hardware accelerator. This leads to a lower activity factor as hardware resources inside the hardware accelerator must wait for partial data transmission to complete. Taking the time for data transmission into consideration, the latency per inference doubled compared to latency without data transmission. As no interrupt has been implemented in the current design, polling is one reason for long data transmission times. Also note that low power domain (LPD) is utilized for the interactions between PS and PL part of the Kria board. Using full power domain (FPD) instead might result in faster data transmission and therefore faster overall inference calculation.

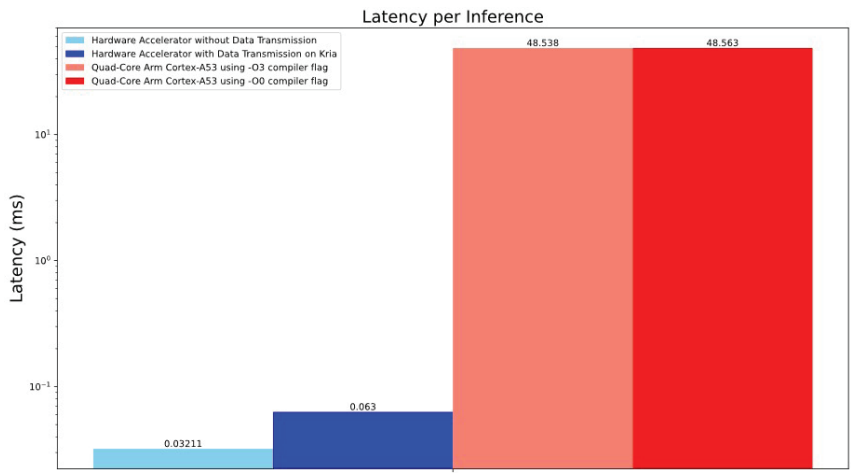


Figure 9.10 Benchmarking Regarding Latency.

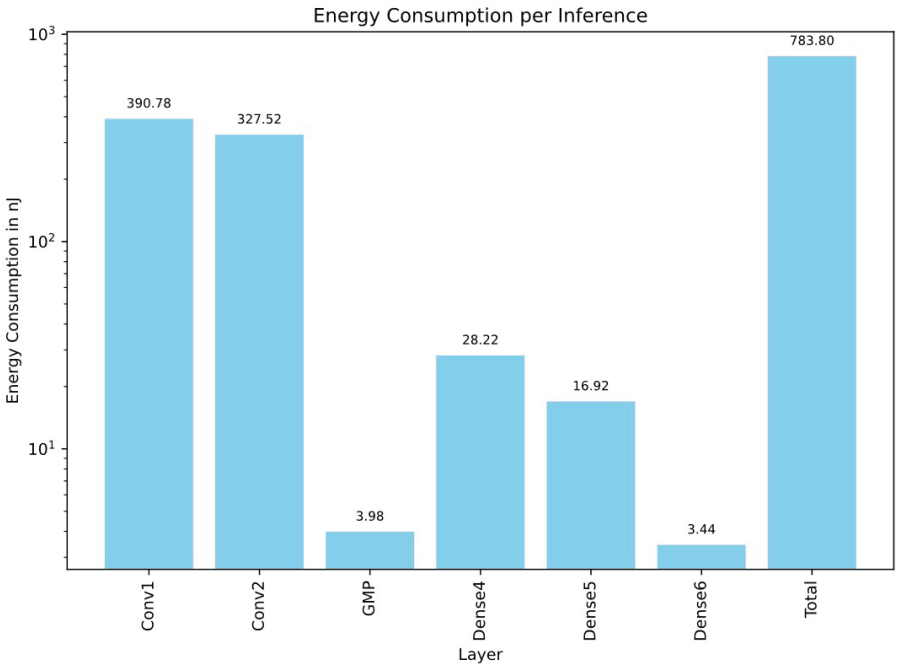


Figure 9.11 Energy per Inference.

9.5.3 Energy

To estimate the energy for one inference calculation required by the hardware accelerator the toggle activity of the design is estimated using Cadence xCelium$^{\mathrm{TM}}$. Furthermore, the design is synthesized using Cadence Genus$^{\mathrm{TM}}$. Based on the toggle activity and the results from place and routing using Cadence Innovus$^{\mathrm{TM}}$, the overall power consumption per inference can be estimated. The energy consumption can then be determined considering the latency information provided in 1.5.2. With more than 91 % of the overall energy consumed, the convolution layers make up for most of the computational effort of the inference. Note that the energy observations in Figure 9.11 do not take data transmission into account.

9.6 Conclusion and Outlook

As demonstrated in previous sections, the hardware accelerator generated for the use case CNN executes inference calculations efficiently regarding latency and energy consumption albeit at the expense of resource utilization.

Considering that more complex image classification tasks require larger CNN architectures [15], the designs of the hardware accelerators generated by tinyHLS need to be minimized in order to meet requirements of complex image classification tasks in the future. As tinyHLS works faster than the AXI4-Lite data transmission, a rational optimization measure to save resources is to execute less operations in parallel. This leads to an increase of latency which remains imperceptible as long as data transmission via AXI4-Lite is slower than data processing by the hardware accelerator. This would also increase the activity factor leading to higher overall efficiency. Another approach to reduce the size of the hardware accelerators is to utilize DSP and BRAM tiles that are commonly available on many FPGAs [16]. Integrating these optimization strategies into the concept of tinyHLS, the translation of larger CNN architectures into reasonable sized hardware accelerators suitable for deployment on standard FPGA boards can be facilitated. For more efficient data transmission, an interrupt signal can be implemented.

As tinyHLS is constrained regarding different CNN architectures, it is essential to develop layer templates for other types of layers such as average-pooling, max-pooling or global-average-pooling in future work. Furthermore, parameters like zero padding or stride can be implemented to provide more flexibility. Average-pooling and max-pooling layers as well as stride are particularly important as they reduce the overall computational effort of the inference calculation.

Acknowledgements

This research has been funded in part by the project CLEVER (Project ID 101097560), which is supported by the Key Digital Technologies Joint Undertaking and its members (including top-up funding by the German Federal Ministry of Education and Research (BMBF)).

This manuscript was edited with the assistance of OpenAI's language model to enhance clarity, readability, and consistency. All suggestions were critically reviewed and accepted or modified by the authors, who take full responsibility for the final content. No AI tools were used to create, alter, or manipulate original research data or results.

References

[1] AMD, AMD Vivado^{TM} High-Level-Design. [Online]. Available: https://www.amd.com/de/products/software/adaptive-socs-and-fpgas/vivado/high-level-design.html (accessed: Sep. 19 2025).

[2] D. Baptista, S. Abreu, F. Freitas, R. Vasconcelos, and F. Morgado-Dias, "A survey of software and hardware use in artificial neural networks," (in En;en), Neural Comput & Applic, vol. 23, 3-4, pp. 591–599, 2013, doi: 10.1007/s00521-013-1406-y.

[3] M. Imani, R. Garcia, S. Gupta, and T. Rosing, "Hardware-Software Co-design to Accelerate Neural Network Applications," J. Emerg. Technol. Comput. Syst., vol. 15, no. 2, pp. 1–18, 2019, doi: 10.1145/3304086.

[4] D. Milojicic, "Accelerators for Artificial Intelligence and High-Performance Computing," Computer, vol. 53, no. 2, pp. 14–22, 2020, doi: 10.1109/mc.2019.2954056.

[5] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays, New York, NY, USA, 2006, pp. 21–30.

[6] C. Qian, L. Einhaus, and G. Schiele, "ElasticAI-Creator," in Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems, New York, NY, USA, 2022, pp. 941–946.

[7] I. Hoyer, A. Utz, C. Hoog Antink, and K. Seidl, "tinyHLS: a novel open source high level synthesis tool targeting hardware accelerators for artificial neural network inference," Physiol. Meas., vol. 13, no. 1, p. 15002, 2025, doi: 10.1088/1361-6579/ada8f0.

[8] AXI4-Lite Interface MIPI D-PHY LogiCORE IP Product Guide (PG202) Reader AMD Technical Information Portal. [Online]. Available: https://docs.amd.com/r/en-US/pg202-mipi-dphy/AXI4-Lite-Interface (accessed: Sep. 19 2025).

[9] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer, "A Survey of Quantization Methods for Efficient Neural Network Inference," in Low-Power Computer Vision: Chapman and Hall/CRC, 2022, pp. 291–326. [Online]. Available: https://www.taylorfrancis.com/chapters/edit/10.1201/9781003162810-13/survey-quantization-methods-efficient-neural-network-inference-amir-gholami-sehoon-kim-zhen-dong-zhewei-yao-michael-mahoney-kurt-keutzer

[10] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A White Paper on Neural Network Quantization," Jun. 2021. [Online]. Available: http://arxiv.org/pdf/2106.08295v1

[11] L. Wei, Z. Ma, C. Yang, and Q. Yao, "Advances in the Neural Network Quantization: A Comprehensive Review," Applied Sciences, vol. 14, no. 17, p. 7445, 2024, doi: 10.3390/app14177445.

[12] TensorFlow, Keras: The high-level API for TensorFlow | TensorFlow Core. [Online]. Available: https://www.tensorflow.org/guide/keras (accessed: Sep. 19 2025).

[13] AMD, AMD KR260 Robotics Starter Kit. [Online]. Available: https://www.amd.com/de/products/system-on-modules/kria/k26/kr260-robotics-starter-kit.html (accessed: Sep. 19 2025).

[14] R. Conlin, K. Erickson, J. Abbate, and E. Kolemen, "Keras2c: A library for converting Keras neural networks to real-time compatible C," Engineering Applications of Artificial Intelligence, vol. 100, p. 104182, 2021, doi: 10.1016/j.engappai.2021.104182.

[15] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, "A Review of Yolo Algorithm Developments," Procedia Computer Science, vol. 199, pp. 1066–1073, 2022, doi: 10.1016/j.procs.2022.01.135.

[16] P. Goswami and D. Bhatia, "Floorplanning of Partially Reconfigurable Design on Heterogeneous FPGA," in Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, 2016, p. 275.