ICT-56-2020 - Next Generation Internet of Things

D 3.4
Final report on the DL accelerator design

Document information	
Contract number	957197
Project website	www.vedliot.eu
Dissemination Level	PU
Nature	R
Contractual Deadline	31.01.2024
Author	Pedro Trancoso (CHALMERS)
Contributors	Muhammad Waqar Azhar (CHALMERS), Fareed Mohammad Qararyah (CHALMERS), Stavroula Zouzoula (CHALMERS), Mario Porrmann (UOS), Marco Tassemeier (UOS), Yu Li (UOS), Marc Rothmann (UOS), Daniel Ödman (EMBEDL), René Griessl (UNIBI)
Reviewers	Daniel Ödman (EMBEDL), Stefan Krupop (CHR)

The VEDLIoT project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957197.

Changelog		
v0.1	22.12.2023	First draft with content from author and contributors.
v0.2	10.01.2024	Draft revised by author and contributors. Version provided for internal revision to the reviewers.
v0.3	22.01.2024	Updated draft and feedback from reviewers incorporated.
v1.0	25.01.2024	Version ready for submission.

Table of contents

Executive Summary

This deliverable describes the design and evaluation of the different FPGA-based Deep Learning (DL) accelerators developed in the VEDLIoT project. These accelerators are the result of Tasks 3.3 and 3.4. To help in the design of these accelerators and also in order to understand their performance better, Task 3.5 focused on the memory hierarchy for the accelerators.

The accelerators presented here were integrated into the VEDLIoT platform developed in WP4. In addition, we followed a co-design methodology for the optimization of certain accelerators. In these cases, models optimized by the tools developed in WP6 were mapped to the accelerators developed in Task 3.4. The evaluation results were then fed back to the optimization tool for further iteration loops to achieve better efficiency while maintaining the required accuracy.

1 Introduction

The advances in both hardware and Machine Learning (ML) algorithms are enablers today for many new applications. Decision making using Deep Learning (DL) models has an increasing demand and while traditionally these applications require large and powerful computer systems, the trend is to offer these same applications at the edge, i.e. closer to where the data is being produced. The implication of this is that accelerators need to provide the required compute power within constrained power and energy budgets. As such, we are increasingly interested in highly efficient IoT accelerators for DL applications. This topic was precisely the focus of the work in Work Package 3, where different types of accelerators were developed. In particular, in this project, we started by analyzing the existing accelerators. The result of this analysis has been reported in Deliverables D3.1, D3.2 and D3.3. In this deliverable, we focus on more flexible and efficient FPGA-based accelerators that were developed by the partners in this Work Package. This is a report of the results from the work in Tasks T3.3 (Design of a reconfigurable DL accelerator), T3.4 (Design of a HW-SW co-designed scalable DL accelerator), and T3.5 (Memory architecture for the DL accelerators).

The work in Task T3.3 focused on developing reconfigurable accelerators based on existing tools offered by the FPGA vendors for this purpose. The main effort in this task was the development of AMD's Deep Learning Processing Unit (DPU) based accelerators for the DL models used in the projects. In Task T3.4, we focused on the development of more dedicated accelerators, and thus, we focused on applying the co-design methodology to develop hardware that matches the requirements of the particular DL models studied in this project. The efforts in this task focused on the development of a complete framework for the design of dataflow DL accelerators (STANN)[25] and the development of a hybrid accelerator that can keep most of the benefits of the dataflow design within a limited hardware resource budget (FiBHA)[1]. The work in Task T3.4 was done in close collaboration with the partners developing the DL model optimization tool in WP6. Together, we developed the accelerators presented here in a close iterative loop, mapping the model to the accelerator, measuring the performance, and giving input for further model optimizations. With this process, we achieved higher efficiency within the required accuracy.

To complement the work of Tasks T3.3 and T3.4, which focused on the compute modules for the accelerator, in Task T3.5, we analyzed the memory requirements that these applications impose on the accelerators. Furthermore, we developed a tool to aid the design, configuration, and later the runtime management of the compute resources, focusing on improving the execution efficiency. This tool provides an easy-to-use graphical user interface to help designers and implementers evaluate different design points for the accelerators so that the best matching accelerator is selected for the required task. Nevertheless, in many cases the workload changes at runtime and thus to further improve efficiency it is necessary to allocate the resources in a better way to match the requirements at that particular time thus saving on unnecessary waste.

The accelerators developed in this work package were integrated into the VEDLIoT IoT platform developed in WP4.

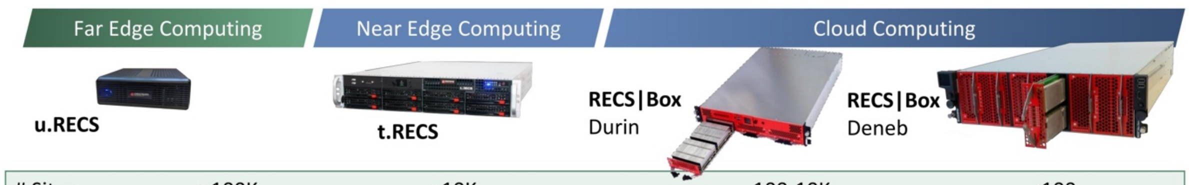
This report is organized as follows:

- Chapter 2 briefly describes the hardware platform used to deploy the accelerators.
- Chapter 3 presents the different DL accelerators:
    - Heterogenous accelerators in Section 3.1,
    - Reconfigurable accelerators in Section 3.2 include both static and dynamic reconfigurable accelerators.

    - The STANN co-design accelerators in Section 3.3.
    - The FiBHA co-design accelerator in Section 3.4.
    - The co-design of accelerators and models with feedback to and from the toolkit developed in WP6 in Section 3.5.
- Chapter 4 presents the memory analysis tool in Section 4.1, a sample analysis that can be done using this tool in Section 4.2, the graphical user interface developed for easy testing of different design points in Section 4.3, and the resource allocation runtime in Section 4.4.
- Chapter 5 summarizes the conclusions for the work presented in this document.
- Chapter 6 presents the list of the bibliographic references used in the document.

2 Hardware platform

VEDLIoT's hardware platforms constitute a collaborative infrastructure specifically crafted to support the diverse developments within the project. This includes a wide range of AIoT applications facilitated by a flexible communication infrastructure and exchangeable microservers. The comprehensive nature of these platforms is illustrated in Figure 1, showcasing the RECS platforms that span application domains from embedded/far-edge computing to cloud computing. Notably, all platforms share a common emphasis on heterogeneous computing, employing tightly coupled microservers.



# Sites	>100K	>10K	100-10K	<100
Footprint	Custom	Compact (1RU)	Medium (2RU)	Large (3RU)
Power Budget	<30 W	< 500 W	500 W – 2 KW	> 2 KW
# Microserver	max 2	up to 3	up to 48	up to 144

Figure 1: Overview of modular and scalable RECS platforms

The cloud computing platform, RECS|Box, stands out with its configuration of two or three rack units strategically designed for high-density applications. This involves the utilization of hundreds of microservers with demanding high-bandwidth communication requirements. In contrast, t.RECS, housed in a single rack unit, is finely tuned for edge computing scenarios with low-latency demands. This is exemplified by specific use-cases such as VEDLIoT's SmartMirror or 5G base stations in the automotive context. Expanding the RECS family, u.RECS extends the range towards low-power and compact embedded computing.

For more detailed information on these hardware platforms, interested parties are encouraged to refer to Deliverable D4.4. This document provides a deeper insight into the intricacies of the platforms, including updates made to RECS|Box and t.RECS with new compute architectures like NVIDIA Orin modules. Additionally, it sheds light on the introduction of u.RECS as a new outcome within the VEDLIoT project, contributing to the project's overall hardware diversity and capability.

3 Deep learning accelerators

In this Chapter, we start in Section 3.1 by presenting the performance of different off-the-shelf DL accelerators that can be integrated into the VEDLIoT platforms. In Section 3.2, we present the FPGA-based reconfigurable accelerators that are built from generic engines (Xilinx DPUs). In that Section, we present not only the static reconfigurable accelerators but also the accelerators that support dynamic reconfiguration at runtime to support different modes of execution, for example.

While generic engine-based accelerators perform well for most DL models, custom-designed accelerators achieve higher efficiency for heterogeneous and dedicated non-standard DL models. Consequently, in this project, we developed different "families" of accelerators covering various design points within the design space to suit different needs and requirements. In the next sections, we will present these accelerators in detail, starting with STANN in Section 3.3 and FiBHA in Section 3.4. We exploit the potential for co-design even further by following a methodology where feedback from the hardware is used as input to the DL toolkit, producing more optimized DL models that are then again used to create improved FPGA-based accelerators. We present the process done in close collaboration with the toolkit from WP6 in Section 3.5.

3.1 Heterogeneous accelerators

A myriad of accelerators catering to diverse applications is available, spanning from diminutive embedded systems with power allocations in the milliwatt range to formidable cloud platforms consuming over 400 W. Figure 2 presents a comprehensive overview of these accelerators on a double logarithmic plot, categorizing them into three distinct groups based on their peak performance values measured in Giga Operations per Second. It is essential to note that the values provided by vendors are utilized without normalization concerning technology, precision, or architecture. On average, these accelerators achieve an energy efficiency of approximately 1 Tera Operation per Watt (1 TOPS/W). The subsequent paragraphs delve into the key characteristics of the three performance groups.

In addition to the diverse landscape of accelerators shown here, it is noteworthy that the VEDLIoT project conducted comprehensive benchmarks on all available architectures. The detailed results of these extensive benchmarks are presented in Deliverable D3.3.

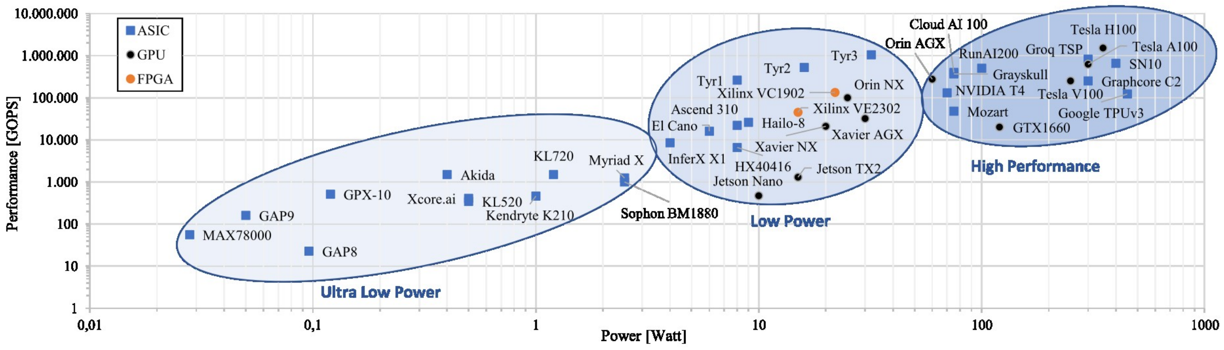


Figure 2: Peak performance of AI Accelerators and classification into performance groups

3.1.1 Ultra Low Power

Accelerators in the ultra-low power group predominantly integrate energy-efficient, microcontroller-style cores alongside compact accelerators tailored for deep learning (DL)-specific functions. Their focus is on generic Internet of Things (IoT) applications, exemplified by devices such as Maxim MAX78000, Ambient Scientific GPX-10, and BrainChip Akida, which provide simple analog or digital interfaces. Other devices, including Greenwave GAP8 and GAP9 Canaan Kendryte K210, and Kneron KL530 and KL720, target vision processing and offer an additional camera interface. Typically, these devices are directly incorporated into applications, eschewing a modular or microserver-based approach due to integrated interfaces and peripherals. Only Bitmain Sophon BM1880 and Intel Myriad X provide a generic USB interface and are designed to function as accelerator devices attached to a standard host processor. The benchmarking activity includes the Intel Myriad X device due to its widespread availability.

3.1.2 Low Power

In contrast to the ultra-low power group focusing on applications with minimal power envelopes, often in battery-powered environments with no specific cooling requirements, the low-power group encompasses accelerators suitable for various applications in automation and automotive industries. All devices in this category feature high-speed interfaces for external memories and peripherals, along with high-speed communication capabilities towards other processing devices or host systems, such as PCIe. This group proves highly suitable for a modular, microserver-based approach, as supported by the RECS platform. With exceptions like Hailo-8, FlexLogix InferX X1, and the VSORA Tyr family designed as dedicated accelerators tethered to an external host processor, the majority of devices include powerful, general-purpose application processors capable of running a fully-fledged Linux operating system. In addition to specialized ASICs like Coherent Logix HX40416, Blaize El Cano, or Huawei Ascend,310, this group also encompasses embedded GPUs from NVIDIA, specifically the Jetson family.

3.1.3 High Performance

The high-performance group of accelerators comprises devices with up to 450 W of Thermal Design Power (TDP), suitable for both inference and training use cases. Typically deployed as PCIe extension cards for edge or cloud servers, this category includes classical NVIDIA Tesla GPGPUs (Tesla V100, A100, H100), dedicated ASICs like Groq TSP, SambaNova SN10, Graphcore C2, or Google TPUv3. Additionally, potent inference ASICs like SimpleMachines Mozart, Tenstorrent Grayskull, Qualcomm Cloud AI 100 Chip, or Untether AI RunAI200 are part of this cluster. As a reference, a consumer-class NVIDIA GeForce GTX 1660 GPU is included in the benchmarking. Despite belonging to the embedded NVIDIA Jetson family, the NVIDIA Jetson AGX Orin is also part of this group due to its high-power envelope.

3.2 Reconfigurable accelerators

As a baseline for the evaluation of FPGA-based DL accelerators, we used the DPU (Deep Learning Processing Unit), an IP core by AMD Xilinx. It is a highly configurable compute engine for convolutional neural networks that can be implemented on Xilinx Zynq UltraScale+ [19], Versal [20], and Alveo devices [21]. Although the architecture differs depending on the device used, it generally consists of a single computing engine containing a configurable number of convolution processing elements for parallelizing the computations. In addition to the processing elements, the DPU also contains an instruction scheduler and on-chip buffer controller. The DPUs use quantized models, i.e., models with weights converted into values of lower precision than the original model. The quantization can be done with the Vitis AI toolbox provided by Xilinx. Besides other features, the toolbox enables the quantization of float models to INT8 and post-quantization optimizations. Post-quantization optimization is applied to the model to counteract accuracy loss due to quantization and to increase performance with methods like pruning. The last step of the model preparation is the compilation targeting a specific DPU.

The DPUCZDX8G is the designated DPU for the Zynq UltraScale+ devices. By increasing peak operations per clock cycle, it is possible to get eight different configurations of different size. The configurations are named according to their peak performance, ranging from B512 (512 peak operations per clock cycle) to B4096. To further increase the performance, DPUs with multiple cores can be implemented, improving the theoretical performance. However, the batch size is limited to one per core. The DPU variant for the Versal AI Core devices is the DPUCVDX8G. Its size can be adjusted by increasing the number of used AI engines per batch handler (e.g., C32) and the number of batch handlers (e.g., B1). The total number of used AI engines is calculated by multiplying the two values. Each batch handler has two interfaces for loading and saving images to memory (indicated by L2S2). Hence, the example above results in the configuration C32B1L2S2.

In addition to the parallelism of the convolutions, which has the highest impact on resource and power requirements but also performance, there are additional parameters. These can be used to adjust the usage of BRAM, UltraRAM, and DSP slices to fine-tune the resource usage, but also, the support for different model features like depth-wise convolution or different activation functions can be enabled. A detailed evaluation of the DPU performance and energy efficiency using different FPGAs in the VEDLIoT testbed is presented in Deliverable D3.3 [22].

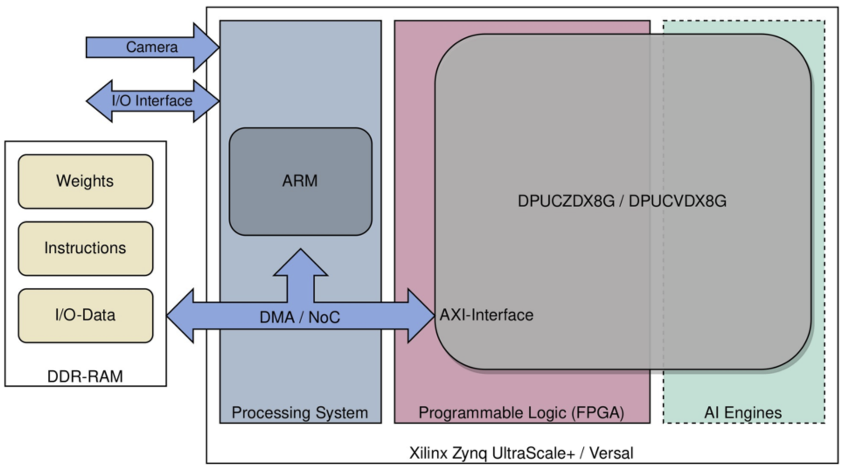


Figure 3: Overview of the internal architecture for using the DPU on a Xilinx Zynq Ultrascale+ and Versal device

In WP4, an FPGA base architecture has been developed to integrate the DPU accelerators into the different RECS systems. The architecture is automatically generated based on a high-level specification combined with a scripting environment. Details about the architecture and design flow can be found in Deliverable D4.6 [23]. From a high-level perspective, the FPGA infrastructure for the Zynq UltraScale+ and the Versal-based DPUs are similar. As shown in Figure 3, both device families comprise an ARM processing system and an FPGA fabric (programmable logic). The Versal architecture provides an additional array of AI engines. Hence, the UltraScale+ DPU uses only programmable logic resources, while the Versal also uses resources from the AI engines. The processing system manages the execution of the machine learning models on the DPU and places the input data in the system memory. The source of the data depends on the system environment. It can, e.g., be obtained by a camera directly connected to the system, received from another device over an interface like Ethernet or PCIe, or loaded from a mass storage device. Besides the input data, the processing system loads the weights and instructions generated by the compilation into the DDR-RAM. The addresses of the input data, weights, and instructions are sent to the DPU through an AXI interface. Finally, the inference is started by the processing system. The DPU can access all the necessary data through its AXI interfaces, in the case of an UltraScale+ device through a DMA interface in the processing system, and in the case of a Versal device using a network on chip, entirely bypassing the processing system. At the end of one inference, the results are stored back to the DDR-RAM, which is where the processing system can read them.

3.2.1 Evaluation of the Xilinx DPU for the VEDLIoT use cases

In addition to the detailed benchmarking of the DPU, discussed in Deliverable D3.3 [22], we have evaluated the energy efficiency and accuracy of the architecture for the VEDLIoT use cases, especially the Smart Mirror (cf. Deliverable D7.5 [24]). Three platforms are used for the evaluation. As an example of a state-of-the-art FPGA architecture, a Xilinx Versal UltraScale+ XCZU15EG MPSoC, integrated on a Trenz TE0808-04 module with a TEBF0808-04A baseboard, is used. Second, a Xilinx Versal XCVC1902 ACAP, integrated into a VCK190 evaluation kit, and finally a 2048-core NVIDIA Jetson AGX Orin 64GB is used as a GPU reference. All implementations on FPGA, as well as on GPU, are based on 8-bit integers. The Xilinx toolflow for quantization and implementation has been used for Versal and UltraScale+, while the GPU results are based on TensorRT. In this example, the mean average precision mAP (.50:.95) decreases from 73.7% to 69.4% when moving from GPU to the reconfigurable architectures. Table 1 compares the performance and energy efficiency achieved for the three architectures. On the UltraScale+ FPGA, two B4096 DPUs with a clock frequency of 200 MHz are running in parallel. In principle, higher clock frequencies are possible, but power limitations of the used board constrained the design. The Versal design is based on a C32 DPU, using 32 AI engine cores per batch handler. Here, the number of batch handlers is varied between one and six. The implementation runs at a clock frequency of 333 MHz for the programmable logic and 1.25 GHz for the AI engines.

All implementations include the complete PetaLinux-based system environment for reading images, storing results and evaluating the accuracy achieved. The UltraScale+ FPGA is significantly outperformed by the other architectures with respect to performance and energy efficiency. For batch size six, the performance that is achieved with the Xilinx Versal is slightly higher than the performance of the NVIDIA Jetson Orin AGX. Due to the lower power of the Versal, it offers the best energy efficiency. Here, the lowest energy per inference is accomplished with the mid-size DPU implementation using three batch handlers.

Table 1: Performance and energy efficiency of YOLOv4 using Xilinx DPUs on UltraScale+ and Versal compared to a GPU implementation on an NVIDIA Jetson Orin AGX

Architecture	UltraScale+ DPUCZDX8G	Versal DPUCVDX8G			NVIDIA Jetson Orin AGX		
Batch size	1	1	3	6	1	3	6
Performance [Inf./s]	19	58	152	195	151	180	192
Power [W]	24.5	19.58	31.10	43.70	51.19	58.25	60.18
Energy/Inf. [mJ]	1283.0	337.6	204.6	224.1	338.1	323.4	313.8

For typical smart home applications with a live camera feed, like the Smart Mirror, a performance of 30 inferences per second is sufficient. Therefore, even the smallest DPU implementation on the Versal offers more than enough performance. With its 32 AI engines, this configuration can also be implemented on significantly smaller Versal devices, targeting edge applications. This way, the required power can be significantly reduced.

The results for YOLOv4 show the advantage of dedicated AI accelerators that are integrated in the Versal ACAP as well as in the GPU. Even with highly optimized vendor-specific IP cores like the Xilinx DPU, traditional FPGAs (like the UltraScale+) cannot provide competitive performance. On the other hand, the results show that the combination of AI engines and reconfigurable fabric in the heterogeneous Versal can provide performance and energy efficiency comparable to GPUs and, in some cases, even higher. For smaller and more heterogeneous models, on the other hand, specifically tailored FPGA implementations may be beneficial. Therefore, we targeted the development of highly optimized accelerators in VEDLIoT, as discussed in Section 3.3.

3.2.2 Dynamically reconfigurable accelerators

The FPGA-based infrastructure developed in WP4 not only supports easy and efficient integration of accelerators but also provides mechanisms for dynamically exchanging hardware accelerators at runtime. This enables us to adapt the architecture to the specific requirements of an application or to changing environmental conditions. In Deliverable D3.3 [22], we have shown the large design space that has to be explored for a single FPGA with different DPU configurations. Various options can be selected, ranging from minimum power (with limited performance) to maximum performance (with high power). But typically, intermediate implementations are of higher interest than these corner cases. E.g., achieving the highest performance or accuracy for a given power budget or achieving the highest energy efficiency for a given performance goal.

This big design space can not only be utilized at design time; many use cases experience different conditions during their runtime. An example is the presence or absence of a person or an object in front of the Smart Mirror (cf. [24]). In this case, it may be desirable to use different design points identified during the Co-Design process, e.g., increasing the performance or accuracy as soon as a person is detected. Dynamic reconfiguration of the FPGA design also provides this flexibility at runtime. Changing the accelerator configuration dynamically can decrease power requirements, when the battery is running low to prolong the runtime of the device. As most of the time, nothing will be in front of the Smart Mirror, it can save power costs by choosing a low-power configuration. However, as soon as someone steps in front of the mirror, a low-power configuration will still be able to detect it and switch to a more performant configuration to run its application with sufficient speed and enable additional resources for new functionalities like gesture detection.

A crucial aspect of runtime reconfiguration is the overhead in terms of hardware resources and reconfiguration time. In VEDLIoT, the overhead accounted for less than 1% of the FPGA resources even on the smallest FPGAs that are used in the testbed (UltraScale+ ZU3EG). In addition, fast DPU reconfiguration in the range of 20ms to 100ms enables smooth switching between low-power and high-performance modes. Dynamic reconfiguration between different DPU configurations could be successfully demonstrated for the different DPU configurations.

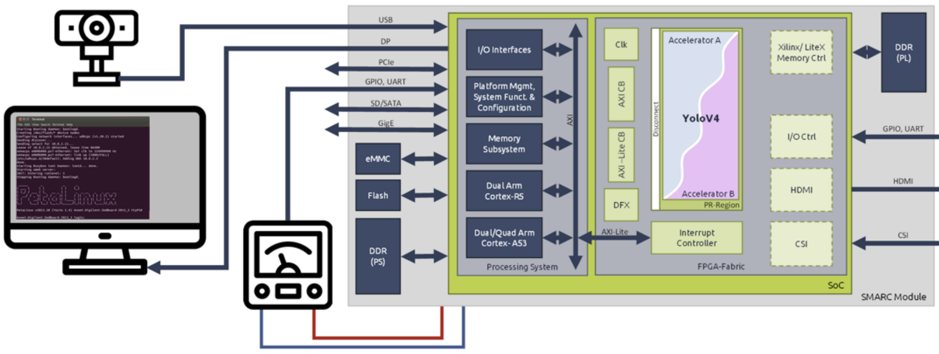


Figure 4: Overview of the internal architecture for using the DPU on a Xilinx Zynq Ultrascale+ and Versal device

Figure 4 illustrates the demo setup. Object detection using YoloV4 is performed with different power/performance trade-offs. The input video stream is first preprocessed. Subsequently, the inference is performed and an output video stream is generated, displaying the detection as an overly in the original video. In parallel, the power consumption of the system is continuously measured, displayed on screen, and recorded for later analysis. At runtime, the user can switch between different DPUs, resulting in a partial reconfiguration of the FPGA, which finally leads to a change in performance and power, which is directly visible in the graphical user interface.

3.3 STANN co-design accelerator

STANN is a library of C++ templates for high-level synthesis that can be used to build hardware accelerators for neural networks. Details about the approach and benchmarking results can be found in [25]. Its main distinguishing feature, compared to existing libraries like FINN [26] or hls4ml [27], is that it supports neural network training while existing libraries focus on accelerating inference. The motivation behind implementing the training of neural networks in hardware is the intention to use STANN in deep reinforcement learning.

The STANN library consists of various C++ template functions necessary for neural network inference and training. These include the forward and backward path of different layer types, different activations, loss functions and their derivatives for training, and weight optimizers like stochastic gradient descent. The template parameters of these functions allow to change the hyper-parameters of each layer (like the number of neurons in a fully connected layer). Additionally, some of these functions have template parameters that influence the hardware architecture that will be generated. For example, the number of processing elements (i.e., the number of multiply-accumulates) used for the matrix multiplication in fully connected and convolutional layers can be configured. It is also possible to choose between different data types for the implementation, including different float and integer formats.

Just calling these template functions sequentially and using the HLS dataflow optimization results in a pipelined dataflow architecture with one hardware module for each neural network layer, as shown in Figure 5. When desired, it is also possible to use STANN to build architectures that reuse hardware for multiple layers of a neural network.

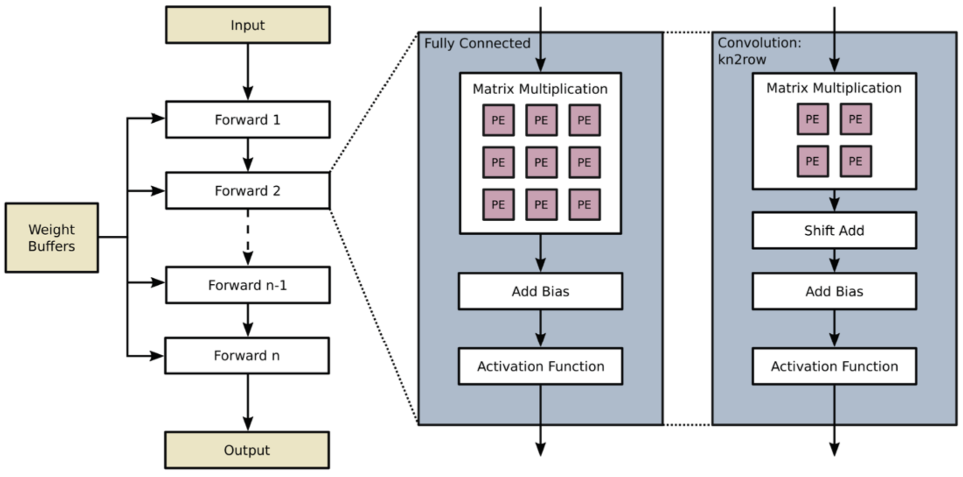


Figure 5: Inference of a neural network using STANN. The white hardware modules are the forward functions of the respective layers, e.g., a fully connected layer or a convolutional layer.

Figure 5 also shows the different steps in a fully connected layer inference, which are implemented in a pipelined dataflow architecture. The most significant computation of the fully connected layer is the matrix multiplication. Matrix multiplication in STANN is implemented as block matrix multiplication, where the matrices are divided into smaller blocks, and the small blocks are multiplied via systolic arrays. The size of these blocks and, consequently, the size of the systolic arrays can be configured with template parameters. This way, the number of processing elements used for the fully connected layer can be configured.

STANN includes three implementations of convolutional layers: direct convolution, im2row, and kn2row, where the last two are used to lower convolution to matrix multiplication [28]. The im2row and kn2row implementations use the same block matrix multiplication as the fully connected layer, enabling the same hardware parameterization. Hyper-parameters like kernel size and number of filters can be configured as well.

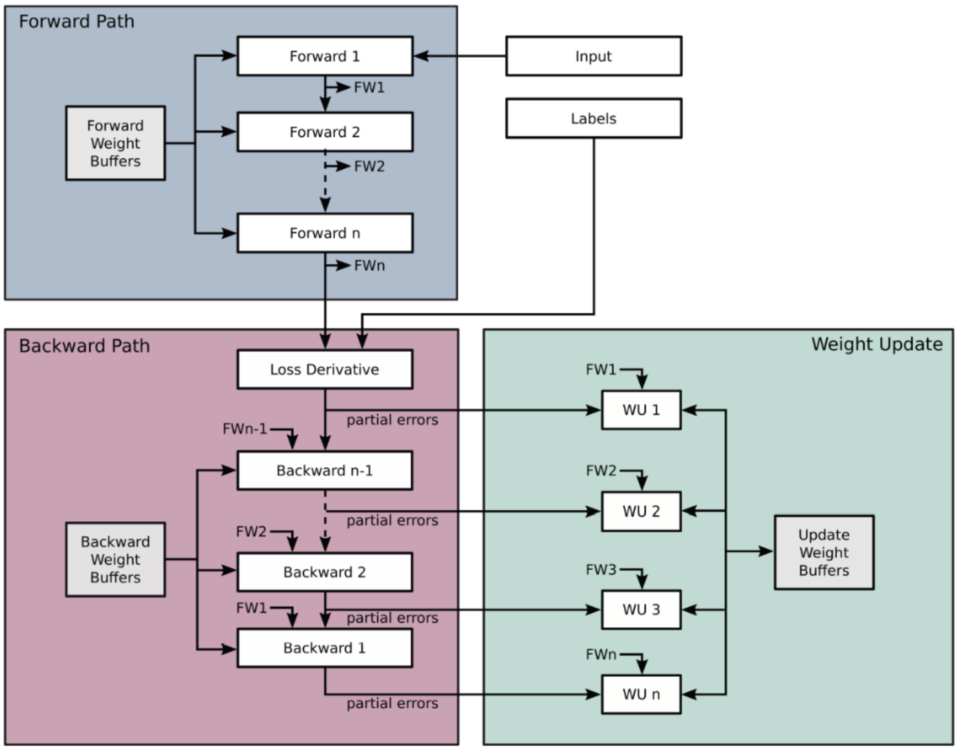


Figure 6: An example of the training process based on a dataflow architecture. The forward and backward hardware modules are pipelined, and the weight update units can work in parallel with the backward path computations.

As for the inference, the training process can be implemented with different hardware architectures in STANN, like a pipelined dataflow architecture or architectures that reuse hardware for multiple layers. A dataflow implementation is shown in Figure 6. The forward and backward modules for each layer can be pipelined, and the weight update can happen in parallel with the backward path. While this architecture is fast, it results in high memory usage because the dataflow optimization of the high-level synthesis requires that each memory can only be read and written once. Thus, the network weights need to be stored three times. This memory overhead is still manageable for small neural networks like they are often used in deep reinforcement learning. It is also possible to use STANN to implement a training architecture that is slower but reuses hardware for multiple layers and does not need these additional copies of the network parameters.

3.3.1 Using STANN for Inference

In addition to benchmarks of STANN that are published in [25], STANN has been used for the implementation of a dedicated DL model specially designed for the Arc Detection application of the Industrial Control and Safety use-case in VEDLIoT, which is discussed in detail in Deliverable D7.5 [24]. The algorithm consists of three steps: data reception, data pre-processing and a neural network for classification. The sampling rate is set to 16 KHz, and 160 data points are transmitted every 10ms to the FPGA. After the data is received, features from the frequency domain are extracted with Fast Fourier Transform. With three hidden layers, the neural network is quite small, but the application has high accuracy and real-time requirements. The STANN-based FPGA implementation has been implemented as an alternative to the GPU implementation, discussed in detail in D7.5 [24]. A major goal is to find a good price/performance ratio. Therefore, we focused on a compact implementation using a small FPGA.

In a co-design approach, both the hardware architecture and the algorithm have been optimized to best match the requirements of the reconfigurable hardware. The incoming 160 raw data points are divided into two groups: the first 128 points and the last 32 points. Each group of data is processed using the FFT algorithm. Next, the two groups of results after FFT, together with the 160 raw points, are normalized and concatenated. Finally, these 320 concatenated data points are used as input features for the deep learning model.

The deep learning model is built using STANN by calling the forward function of the layers sequentially with a dataflow architecture. The dataflow architecture allows for the highest throughput and lowest latency. Four fully connected layers are connected via HLS streams. The dataflow optimizations are applied to the network function to pipeline the network between layers. The number of processing elements (PEs) per layer can be chosen by configuring the block size for the matrix multiplication.

The hardware platform for which the STANN-based implementations are evaluated is an Avnet Ultra96-V2 board equipped with a Xilinx Zynq UltraScale+ ZU3EG FPGA. All the implementations in this evaluation are synthesized and tested with a 250 MHz clock frequency. The data processing block is realized based on floating-point data for all the implementations. Furthermore, the deep learning model is implemented in two data types, namely floating-point and half-precision floating-point. For each data type, the number of PEs per layer can be changed by configuring the block size for the matrix multiplication. For example, a single processing element can be used for each layer, which will lead to high latency but will also occupy the least resources. Alternatively, multiple processing elements can be used for each layer, which results in lower latency by occupying more resources.

Table 2 shows the resource usage and the latency of the accelerator with different data types and PEs per layer. From the given resources, the two FFT implementations together require 2.3k LUTs, and 24 DSP slices, illustrating that most of the resources are used for the neural network accelerator. With respect to latency, the FFTs only add around 1 µs, while most of the time is spent on the NN accelerator.

Table 2 Resource usage and latency of different implementations (FFT and neural network accelerator)

Data Type	PEs/ Layer	BRAM	LUTs	DSPs	Latency	Power	Accuracy
Float	1	161 (74%)	19862 (28%)	76 (21%)	192µs	3.02 W	99.07%
Float	4	161 (74%)	25993 (37%)	100 (28%)	60 µs	3.04 W	99.07%
Half	1	86 (40%)	17338 (25%)	84 (23%)	70 µs	2.92 W	97.66%
Half	4	86 (40%)	20007 (17%)	101 (28%)	42 µs	3.00 W	97.66%

As expected, the implementations with multiple PEs per layer are faster but require more resources and power. For the float data type, the implementation with 4 PEs per layer is more than three times as fast as the implementation with a single processing element per layer. But it also requires about 6100 additional LUTs and 24 more DSP slices. For the half data type, the implementation with 4 PEs per layer is nearly two times faster than the implementation with a single processing element per layer. But it also requires significantly more LUTs and DSP slices. In addition to the resource requirements, Table 2 contains the data about the on-chip power and the accuracy of each implementation. The accuracy is evaluated based on the same test dataset as used for the GPU implementation. All proposed implementations of the arc detection accelerator are fast enough to meet the latency requirements of the application, with the FPGA implementation accomplishing this with slightly lower power than the GPU (up to 7W for the total system power using the GPU compared to 5.9W for the complete FPGA-based system). For the float implementation, a speedup of 183 has been achieved compared to the initial base implementation, which required 11ms for computation on GPU. The energy efficiency was increased by a factor of 186 from 66mWs to 0.35mWs per iteration. Based on the specific needs of a system, one specific implementation can be chosen regarding the amount of resources, accuracy, and power consumption for real-time scenarios.

3.3.2 Using STANN for Reinforcement Learning

The extension of STANN towards reinforcement learning has been successfully applied for the control of electrical drives in the VEDLIoT Open Call Project “Power-Edge-RL”. An FPGA-based accelerator was specifically designed for Deep Q Networks (DQN) for this project and was used by the Open Call partner from Paderborn University. Using a Xilinx Versal, a speedup of 7x was achieved when compared to their previous solution. Details about the implementation can be found in Deliverable D7.8 [29].

3.4 FiBHA: Fixed budget hybrid CNN co-design accelerator

In the quest for resource and energy-efficient Deep Learning (DL) algorithms, state-of-the-art DL models, especially Convolutional Neural Networks (CNNs), are becoming more heterogeneous. The heterogeneity in these CNN models is present at two levels. The first level of heterogeneity, which is common in all CNNs, is heterogeneity in layers’ filters, input and output shapes and sizes, number of operations, and reuse patterns. These variations are present within layers of the same type; hence we refer to it as intra-layer-type heterogeneity. Another form of heterogeneity, which is increasing over time, is a result of including new layer types. For example, compared to AlexNet and VGG which have one form of convolution, recent resource-efficient CNNs like ResNet, XCeption, EfficientNet, and MobileNet have new types including Pointwise (PW) and Depthwise (DW) convolutions. Generic, one-size-fits-all accelerators cannot capture these levels of heterogeneity. As a result, novel model-specific accelerators with dedicated modules or engines are being proposed.

When reviewing the literature, the model-specific accelerators could be categorized into three categories. The first category emphasizes reusability and optimizing for the common case. The accelerators belonging to this category contain a minimal number of dedicated engines such that all the layers of one type (e.g. pointwise convolutions) are processed by one engine. Such accelerators address the inter-layer-type heterogeneity but do not capture the heterogeneity among layers of the same type. The second category of accelerators emphasizes highly dedicated design and full capturing of heterogeneity. This category’s accelerators have one dedicated engine per layer, which makes it resource-demanding and unscalable. FiBHA (Fixed Budget Hybrid CNN Accelerator) is a hybrid architecture that represents a middle point in the design spectrum between the aforementioned two categories [1]. It combines architectural design from both categories such that more heterogeneity is captured than in the first category while being more resource-aware than in the second. FiBHA is implemented and evaluated using high-level synthesis (HLS) on an FPGA (Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit). FiBHA achieves up to 1.7x and 4.1x of the throughput achieved by state-of-the-art accelerators of the two categories using the same resource budget.

3.4.1 Model-Specific CNN accelerators taxonomy

We categorize the custom and model-specific accelerators in literature into three main categories. Figure 7 shows a high-level visualization of these categories. They are:

- **Single Engine Multi Layer (SEML)**: these accelerators process all the layers of a particular type using the same - single - engine. On one hand, these accelerators are relatively resource-efficient as they have a minimal number of engines. On the other hand, processing all layers of one type using the same engine requires that this engine is designed to be efficient for the common case. This results in inefficiencies in processing layers that do not strictly resemble that case. Examples of such accelerators are [2-8].
- **Single Engine Single Layer (SESL)**: these accelerators process each layer using a distinct engine. These accelerators are also referred to as streaming, Synchronous-Dataflow (SDF)-based, or pipelined architectures. On the one hand, these accelerators contain engines dedicated to each layer, which could potentially maximize efficiency. On the other hand, designing one engine per layer is resource-demanding. Moreover, these accelerators are not scalable as CNNs become deeper. Moreover, in an SESL accelerator, all the engines that represent pipeline stages should have the same execution time. Otherwise, there will be pipeline idle stages resulting in suboptimal performance. In a balanced SESL, theoretically, all engines are active all the time, eliminating any resource idleness or underutilization. Achieving such balance given deep CNNs may not be feasible. Examples of such accelerators are [9-12].
- **Hybrid**: this is the accelerator design we proposed in [1]. The idea is to strike a balance between the SEML and SESL categories. The hybrid accelerator is composed of two parts: one part resembles the architecture of a SESL accelerator and another resembles SEML. The SESL-like part is used to process layers that have more heterogeneity. The SEML part is used to process layers that are relatively less heterogeneous.

Notes:

- The term single layer is used in this context to refer to a fused implementation of convolution, batch normalization (BN), and activation (e.g. ReLU) layers. This is because the common practice is to fuse convolutional layers with the batch normalization (BN) and activation layers following them.
- The presented taxonomy focuses on the mapping between the model layers and the engines, rather than the engine count. For example, in a Single-Engine Multi-Layer (SEML), there could be multiple engines, but the emphasis is on the fact that each “single engine” processes multiple layers.

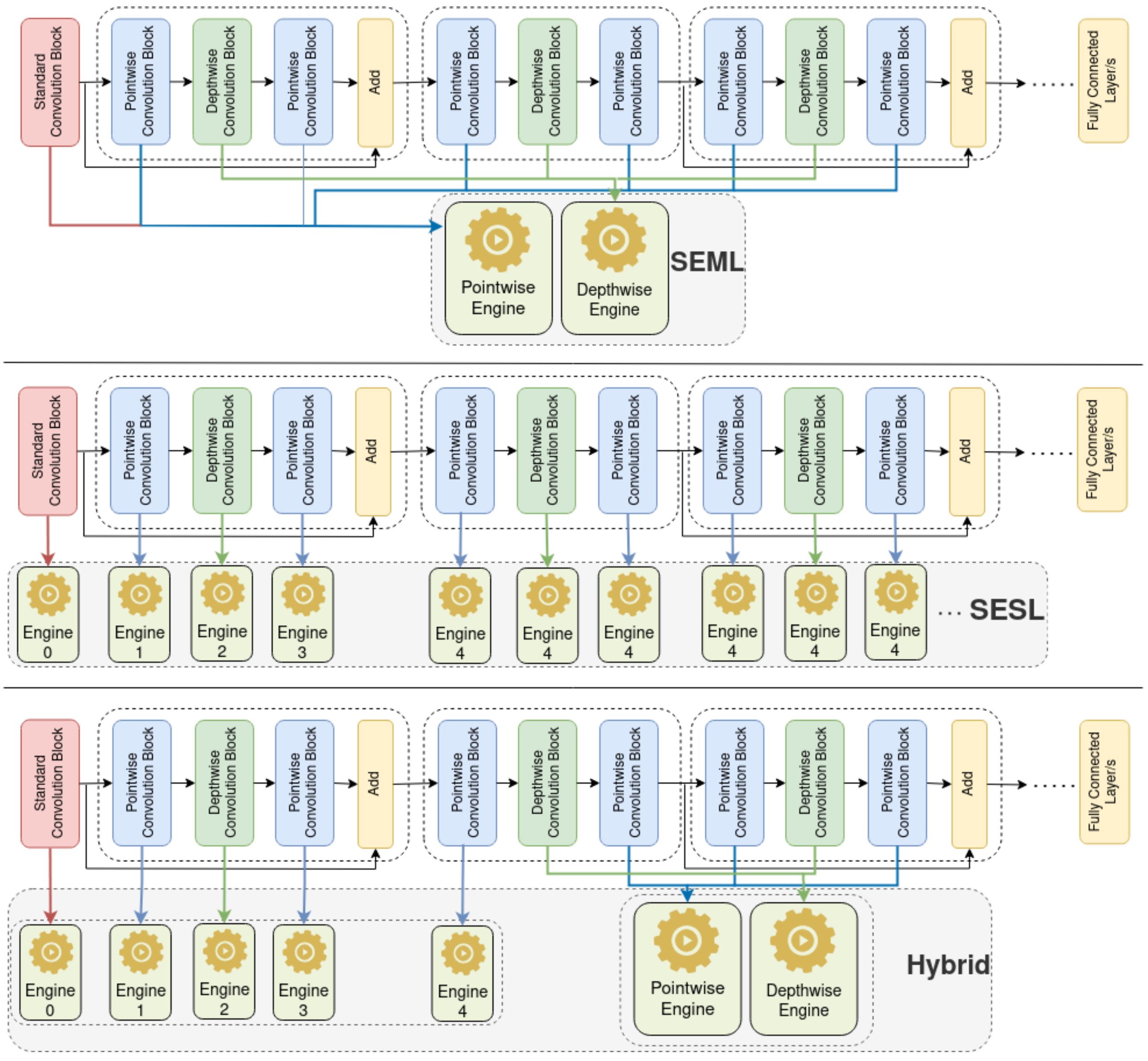
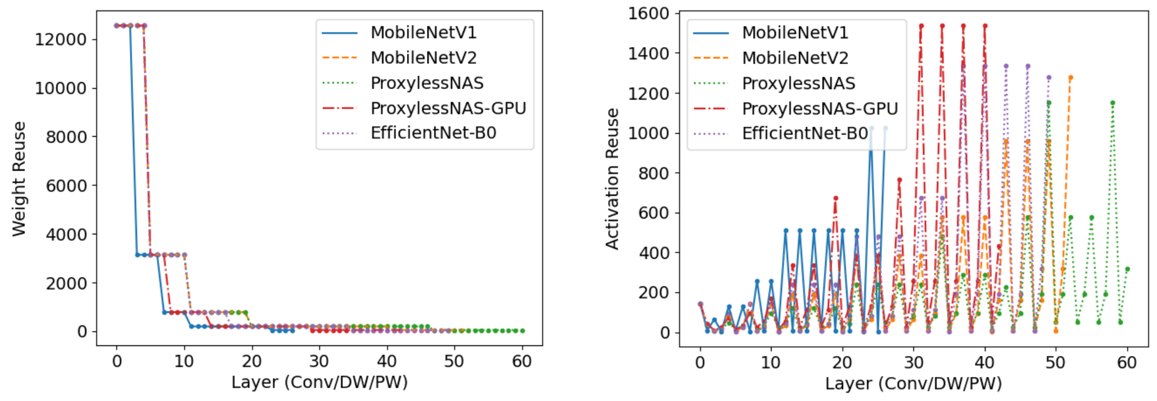


Figure 7: A high-level visualization of the SESL, SEML, and Hybrid accelerator architectures. The visualization focuses on the mapping between the convolutional layers of a CNN and the engines of accelerators, the internal architecture of the engines, their interconnectivity, and their memory system details are omitted for simplicity.

3.4.2 Heterogeneity in resource-efficient CNNs



(a) The number of times weight is reused in a certain layer. *(b) The number of times an activation is reused in a certain layer*

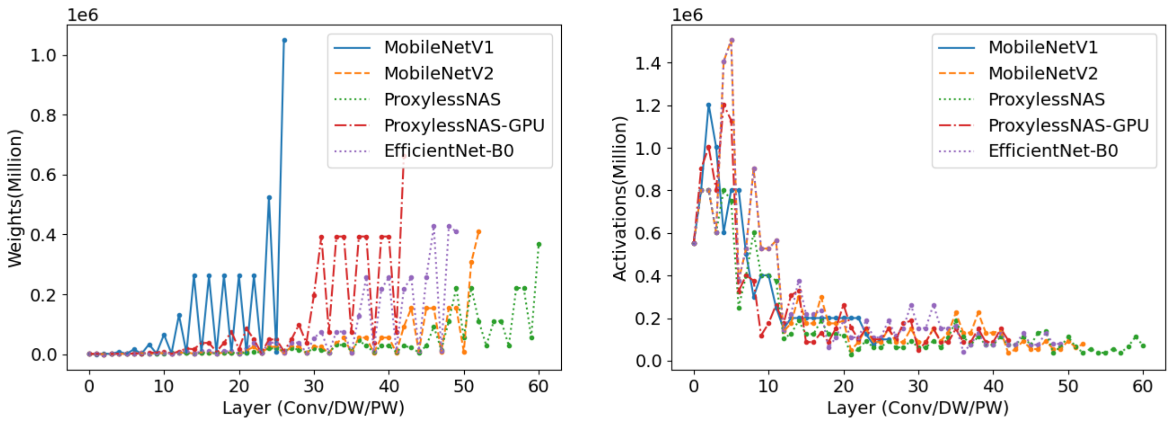
Figure 8

As we briefly described, there are two levels of heterogeneity in resource-efficient CNNs. The first form is inter-layer-type heterogeneity, which is obvious. It is a result of having various convolutional layer types. The sample CNNs shown in Figure 7 show the convolutional layer types that are common in the targeted CNNs which are Standard, Pointwise, and Depthwise. The second level is heterogeneity among layers of the same type. Figure 8 shows some aspects of this heterogeneity in a set of efficient CNNs, namely MobileNetsV1[13], MobileNetV2[14], ProxylessNAS[15], and EfficientNet[16]. Figure 8.a depicts the weight reuse of the layers of these models. The term reuse indicates the number of times a single element (weight/input/output) is used. The reuse is synonymous with the compute-to-memory access ratio. Maximizing reuse is essential for reducing data-movement, hence latency and energy consumption. A weight is reused as many times as the product between the height and the width of the output feature maps (OFMs). Figure 8.a shows that the initial layers have high and rapidly changing weight reuse; the rest of the layers have low and less changing weight reuse. The weight reuse is high in the initial layers because the feature maps (FMs) are wide (have relatively large height and width). Hence, each weight is used with a relatively large number of elements in each of these FMs. However, CNNs gradually reduce the FMs' width and height but expand the number of channels (FMs become narrower but deeper). The weight reuse varies more among the initial layers since they have more frequent sub-sampling resulting in frequent changes in the FM's width and height. However, in the rest of the layers, these dimensions change less frequently; hence the reuse is steadier.

Figure 8.b depicts the input reuse. For DW layers, input reuse is calculated as filter width * filter height. For PW layers, input reuse is equivalent to the number of filters. Input reuse has almost the opposite behavior compared to weight reuse. It is low in the initial layers and high in the rest. Moreover, unlike weight reuse, input reuse fluctuates. This is a result of the differences between the reuse in the DW and PW convolutions. The peak values of the input reuse represent the reuse values of all the PW layers. The low values of the input reuse represent the DW layers. Note that input reuse is negligible in the case of DW layers. Hence, regarding input reuse, the focus is on the higher and more dynamic reuse of the PW layers.

3.4.3 FiBHA design

The main design objective of FiBHA is to handle the two levels of heterogeneity within a fixed resource budget. FiBHA targets meeting the processing elements (PEs) budget available on a specific FPGA board, reducing the memory requirements, and reducing the traffic on the off-chip memory bandwidth. In the rest of this section, we argue that using an SESL-based design to process the initial layers of a CNN and an SEML-based design to process the rest is the option that satisfies FiBHA design objectives.



(a) Weight size per layer (equivalent to the memory needed to store that layer weights assuming 8-bit weights).

(b) FMs size per layer (equivalent to the memory needed to store that layer inputs and outputs assuming 8-bit FMs).

Figure 9

Objective 1: Maximizing the captured heterogeneity

SESL architecture captures the two forms of heterogeneity; hence, it is the right option compared to SEML when there is more heterogeneity. When analyzing CNNs' layers, it is found that the initial layers have more heterogeneity than the rest. The following are three examples of such forms of heterogeneity:

- Weight reuse heterogeneity: Figure 8.a shows that the weight reuse changes dramatically among the initial layers. However, in the rest of the layers, the weight reuse is barely changing.
- Parallelism patterns heterogeneity: The initial layers, especially the first layer, have shallower inputs compared to the rest. This leads to these layers having different parallelism requirements. As a result, when computed using the same engine that is used to process most of the other layers, these initial layers have low resource utilization.
- The first layer in the resource-efficient CNNs is the only standard convolution layer in all the models we have experimented with. This makes it a source of inter-layer-type heterogeneity.

Objective 2: Minimizing the required memory

Figure 9.b depicts layers on the x-axis and the size of FMs on the y-axis. It shows that the initial layers have much larger FMs than the rest. This is common in most of CNNs [1]. This analysis suggests using SESL design to process these layers. Unlike SEML where the outputs need to be fully written to the main memory and then read when the next layer is computed, in the SESL data flows between the pipelined engines and is processed almost immediately. As a result, when SESL is used to process the initial layers, much of the intermediate results (FMs) memory is saved.

Objective 3: Avoiding off-chip memory bandwidth bottleneck

In an SESL architecture, the engines run simultaneously and need to be fed with weights at the same time, putting pressure on the off-chip memory bandwidth and making it a bottleneck. As Figure 9.b shows, the initial layers have small weights that can be stored on-chip. As a result, these layers could be processed with an SESL architecture without the need to use the off-chip memory. On the other hand, the layers towards the end of a model have much larger weights that should be stored off-chip. The increase in the size of the weights as we move from the first towards the last layers happens because the FMs channels increase, requiring more and larger filters.

Based on this analysis, the SESL part is used to process the initial layers, and the SEML is used to compute the rest.

3.4.4 FiBHA Architecture Overview

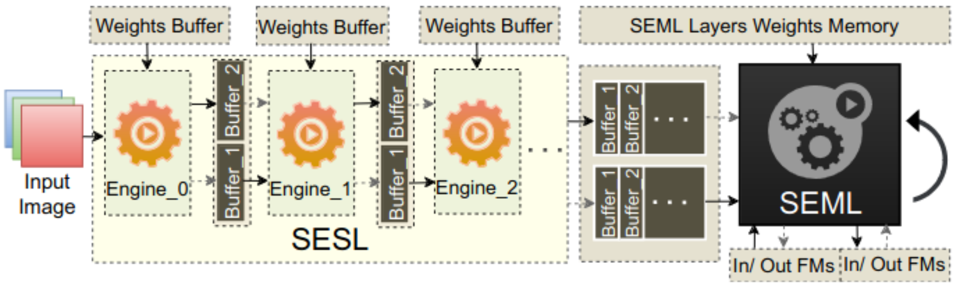


Figure 10: FiBHA architecture: a high-level overview

Figure 10 shows a high-level overview of FiBHA architecture. A FiBHA instance is composed of two main parts: a SESL, and a SEML. Double buffering is used to bridge these two parts such that both can work concurrently. For instance, while the SEML part is processing the input n, the SESL could process the next input (n+1).

In the SESL part, the engines are pipelined to maintain a high throughput. Double buffering is applied between these engines as well. In addition to the buffers between the engines, each engine has its own weight buffer. The input image is processed tile by tile. Assuming we have l engines in the SESL part, while Engine_0 is processing chunk l + n, Engine_1 handles chunk l + n - 1, and so forth. Once Engine_l-1 completes a chunk, it pushes it to one of the two buffers connecting the SESL part to the SEML part. Each of these engines is assigned a number of PEs such that the execution times of the engines are the same so that resources are well-utilized. Layers l+1 up to the last layer are executed using the SEML part. This part could be implemented using any SEML architecture. It processes its layers one by one and has two buffers used in an alternating fashion to load and store the inputs and outputs.

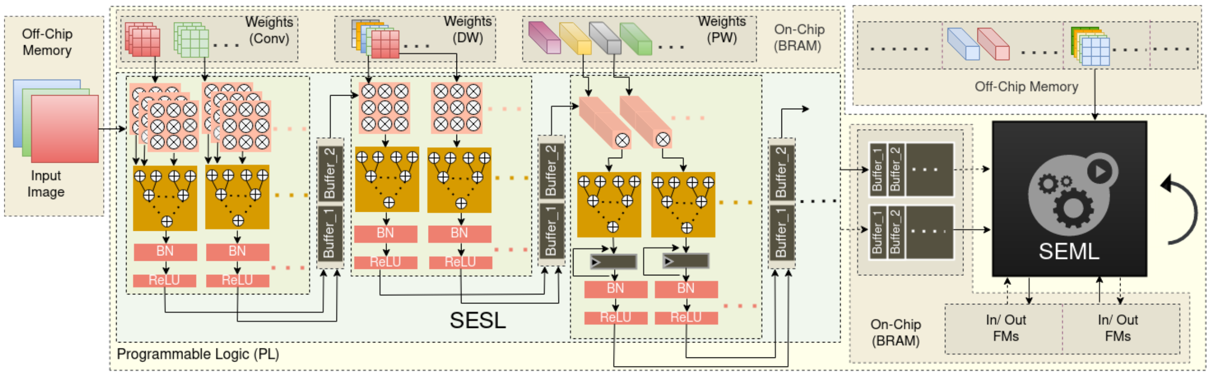


Figure 11: FiBHA: an example of an implementation.

The implementation details of the engines are not the core focus of FiBHA design. However, Figure 11 shows a more detailed example of FiBHA implementation showing a possible implementation of the engines. The figure shows that each of the SESL layers may have its own design and degrees of parallelism. It shows as well that the weights of the SESL layers are stored on-chip to have higher bandwidth, while the weights of the SEML part are stored off-chip as they are usually large. The SEML part is shown as a black box to indicate that any state-of-the-art SEML design could be used to implement that part.

Figure 12 shows an example of the mapping between different CNN layers and FiBHA engines. A detailed description of how the blueprint of a FiBHA instance is derived and how the resources are distributed on the engines given a certain resource budget and CNN characteristics can be found in [1].

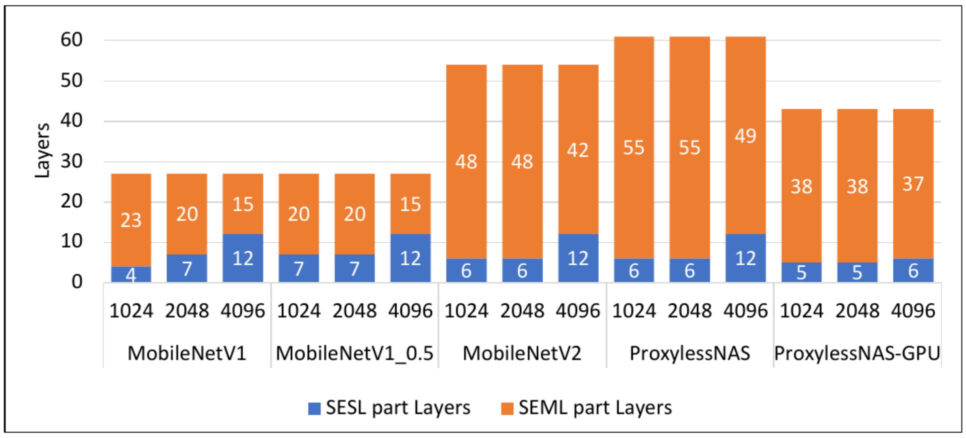


Figure 12: Number of layers assigned to each of FiBHA parts.

3.4.5 Evaluation

3.4.5.1 Experimental Setup

We used Vitis-HLS and Vivado (2021.2) to implement and evaluate our accelerators. The board we targeted is Zynq UltraScale+ MPSoC ZCU102 with an XCZU9EG FPGA and quad-core ARM Cortex-A53. The clock frequency used is 167 MHz for all the synthesized accelerators. The evaluation is done under different PE budgets to simulate different scenarios and evaluate scaling. A resource budget is used to control the combined degrees of parallelism of all the engines of the accelerator under evaluation. For example, in the FiBHA instance that uses a budget of 1024 PEs, the parallelism in the SESL part engines and the SEML part engines combined is less than or equal to 1024.

Baseline Accelerators: We used two accelerators to evaluate our design.

bSEML: We use an SEML accelerator architecture based on the one proposed in [17]. We refer to this accelerator as bSEML. This accelerator was originally used for MobileNets, but we extended it to handle other models in our evaluation set. It handles the DW and PW convolutions using two dedicated engines. More details about this architecture and how we modified it can be found in [1,17]. For FiBHA and SEML, the fully connected layer is implemented using the Cortex-A53 vector extension.

FINN: FINN is an experimental framework from Xilinx Research Labs to explore deep neural network inference on FPGAs. FINN generates model-specific streaming-dataflow accelerators. FINN is an example of an SESL architecture. By specifying a desired throughput, FINN generates an accelerator design that guarantees that throughput, as long as the available resources can support it. Our experiments report the maximum supported throughput by FINN and the corresponding resource utilization. Our comparisons with FINN are limited to MobileNetV1 variants as they are the only models in our set with an available FINN end-to-end open-source implementation.

Evaluated CNNs: We used a representative set of heterogeneous and hardware-efficient CNNs in our evaluation. They are MobileNetsV1 [13], MobileNetV2 [14], ProxylessNAS [15], and EfficientNet [16]. More details about these model architectures are found in the referenced papers.

3.4.5.2 FiBHA vs alternatives

Since FiBHA is proposed as an alternative to the SEML and SESL accelerator categories, we evaluate its performance against two state-of-the-art accelerators belonging to these two categories.

Figure 13 shows that FiBHA outperformed the alternative designs achieving up to 4.1x and 1.7x throughput improvements over FINN and bSEML respectively. The main reason behind outperforming bSEML is the capture of more heterogeneity, especially among the initial layers of CNNs as we have discussed in the previous sections. A more in-depth analysis that quantifies how capturing heterogeneity is translated into performance improvement can be found in [1].

When compared to FINN, FiBHA's advantage is more apparent. One issue with pure SESL architectures, like FINN, is the high resource requirements as each layer has its own engine. Another issue is the need to balance the stages of a pipeline that incorporates engines of versatile workloads. FiBHA avoids both issues by having a relatively shorter pipeline that is both more resource-light and easier to balance.

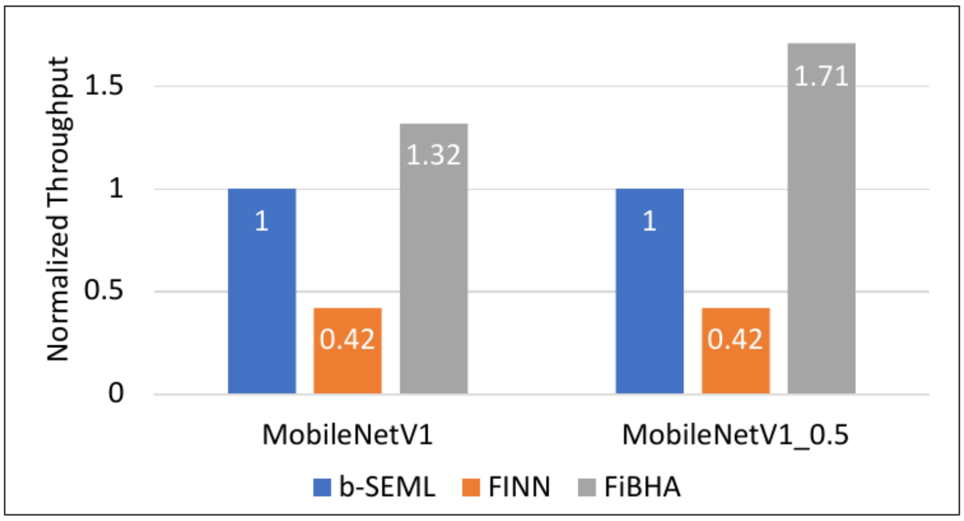


Figure 13: b-SEML, FINN, and FiBHA throughput normalized to b-SEML throughput.

3.4.5.3 FiBHA Scalability: various resource budgets

Figure 14 shows the throughput improvement of FiBHA compared to bSEML using various models and PE budgets. We mainly compare with bSEML when studying scalability as FINN design is not scalable. The figure shows the number of PEs on the x-axis and the estimated and the measured throughput of FiBHA (both normalized to that of bSEML) on the y-axis. The estimated throughput is reported by our FiBHA generation heuristic SplitCNN [1], and the measurement is based on the actual runs. The figure shows that FiBHA scales well and continues to outperform bSEML under different PE budgets. This demonstrates that the hybrid design has an advantage under different PE budgets. Moreover, FiBHA's throughput usually scales better than bSEML. Except for one case, FiBHA either maintains or achieves higher throughput improvement when increasing the number of PEs.

Unlike an SEML design where all the PEs can be mapped to the reusable engines, in FiBHA the PEs have to be split among the SESL and the SEML engines. This gives the SEML part of FiBHA fewer PEs resulting in a lower throughput on that part. However, as the SESL part of FiBHA improves efficiency, the net result is an improvement over an SEML. However, when the PEs number increases, it is more likely that the SEML engines will be underutilized in some layers. In such cases, assigning fewer PEs to FiBHA's SEML part does not cause performance degradation which makes FiBHA's heterogeneity-capturing advantage directly translate into performance improvements.

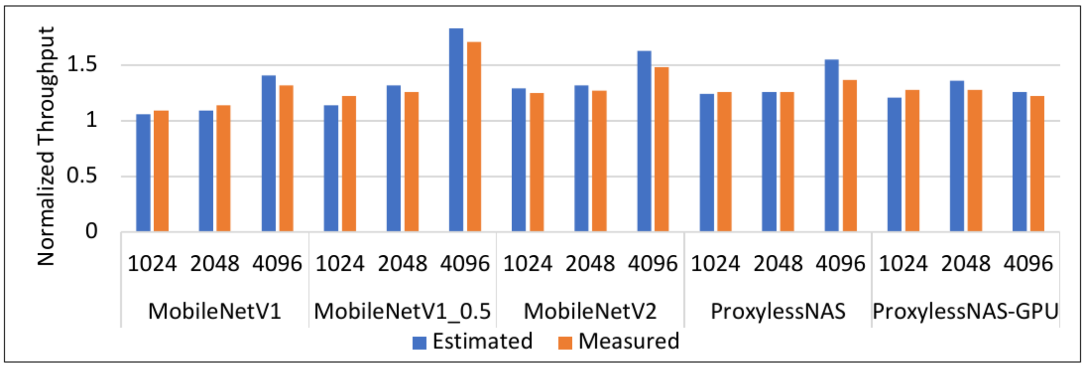


Figure 14: FiBHA estimated and measured throughput normalized to b-SEML throughput

3.5 Co-design of DL accelerators and models with feedback to the toolkit

In the previous sections, we presented a set of accelerators that have been proposed in the context of software-to-hardware co-design. Meaning that these accelerators are model-aware and designed given model characteristics. In this section, we present an effort in the hardware-to-software co-design direction to complete the co-design bidirectional process. We redesign the DL model given the proposed accelerator characteristics. Even though the co-design efforts in these directions are done independently, we argue that with an engineering effort the process could be automated and turned into an online bi-directional co-design loop.

The targeted CNN model in this part of the work is MobileNetV2. MobileNetV2 is selected as a representative resource-efficient CNN model. Unlike traditional CNNs, e.g. VGG or ResNet, MobileNetV2 includes more layer types making it a more interesting and challenging target. Regarding the accelerators, we use both FiBHA and DPU which have different design philosophies; hence they present two interesting points in the design spectrum.

3.5.1 Model Co-design process: goals and steps

The targeted metric of interest in our model co-design process is decreasing inference latency with minimal loss in accuracy. Latency is chosen because we target real-time systems, mainly automotive use cases, where meeting certain latency constraints is crucial.

Pruning is used to reduce models' weights and operations, hence their latency on a given hardware. However, the accuracy-latency trade-off of pruning is hardware-dependent. Applying uniform pruning on the model as a whole prevents exploring the design points that represent the accuracy-latency sweet-spots given our custom accelerators. To extend the search space, we apply non-uniform, as well as uniform, pruning at layer rather than model granularity and analyze the impact of different pruning ratios to derive optimal ones.

As the search space is huge, we use a machine learning-based approach to search for promising pruning ratios. There are two main steps in this search process. The first step is generating a relatively large number of pruned models, running them on the targeted accelerators, and collecting the metrics of interest. The second step is feeding the results into the machine learning model and predicting the pruning ratios that maximize the gains. The result of the second step is a model or model family that is pruned according to the predictions and optimized for specific hardware. In other words, the second step is figuring out the parameters of the hardware-aware pruning.

A latency predictor comprising a neural network is trained to estimate the expected latency of a candidate model on the target hardware. The training of said performance predictor was done by generating a data set consisting of 1000 semi-randomly generated candidate models that were evaluated on the hardware target of choice. The resulting data set is a simple data set with the model architecture as input and the expected latency on hardware as the label for the input sample. This data set is then split up into a training set and a validation set where the validation set is a random subset of 100 models from the 1000 models and the training set is the remaining 900. The model is trained until convergence, typically 1000+ epochs.

Figure 15 shows the characteristics of 1000 of the first-step models that are pruned non-uniformly. It shows both the number of parameters (weights) of these models and their number of operations normalized to those of the original, unpruned, model. Figure 16 shows the number of operations of a set of models belonging to the second step. It shows that there is a difference between the pruning for FiBHA and pruning for DPU, which is aligned with our assumptions that hardware should be considered while pruning. In the following section, we demonstrate that one iteration of this co-design process led to generating models that have considerably lower latency than their unpruned counterparts without a noticeable loss in accuracy.

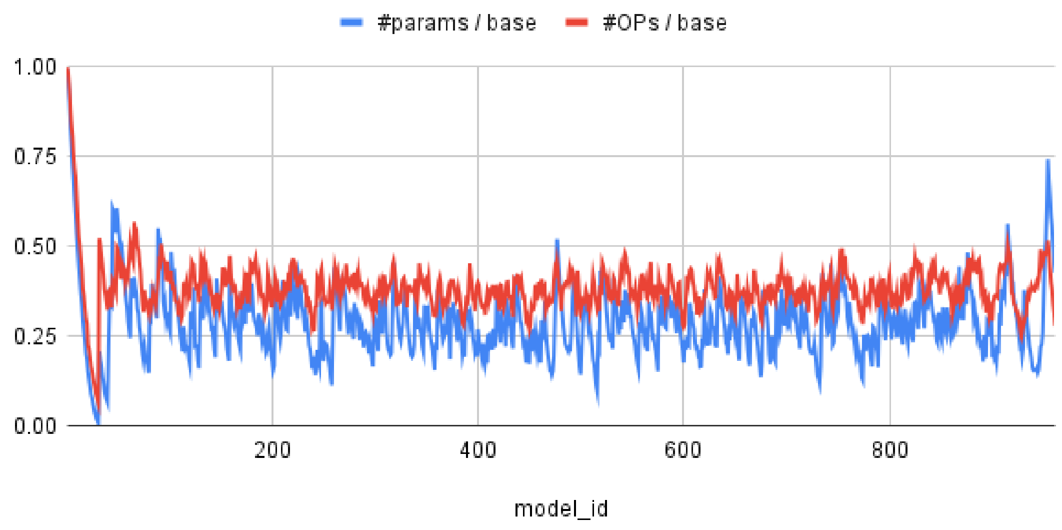


Figure 15: First step models: number of weights and operations of models with layer-wise pruning normalized to those of the original model.

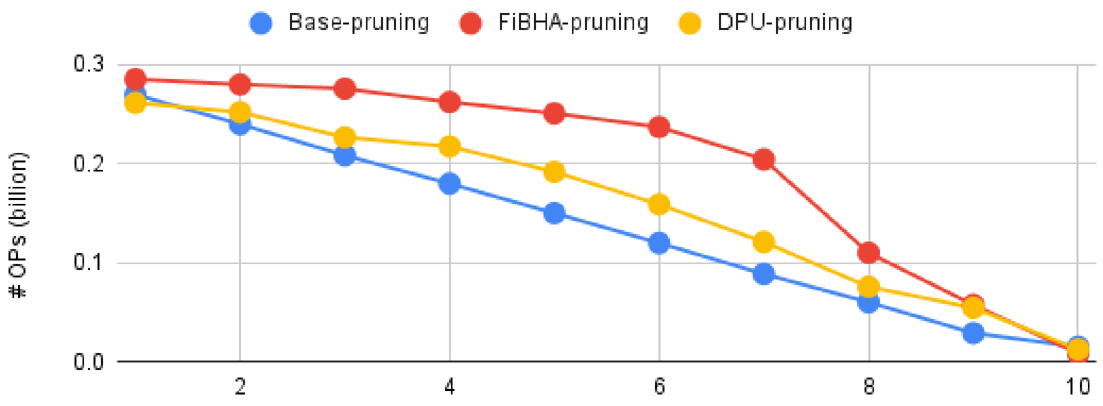


Figure 16: Second step models: number of weights and operations of models with hardware-aware layer-wise pruning normalized to those of the original model.

3.5.2 Model Co-design process: results

In this section, we present the results of a sample of 10 models of the second step that are shown in Figure 16. We use 8-bit integer quantization (INT8). The quantization for FiBHA is done using the Tensorflow-Lite library and for DPU using Vitis-AI.

Figure 17 shows the difference in latency between the predicted latency and the actual latency measured on the HW. For FiBHA, the predicted latency in 9 out of the 10 cases is less than 8% away from the actual value. There is one outlier. Note that this outlier is not crucial. This is because it is an extreme case that has a very low number of weights and operations (figure 16) and has the lowest accuracy. Hence, it does not represent an interesting design point. For DPU, the predictions are even more accurate. The largest difference is roughly 12% and the average is 2.3%. This is probably because the DPU has a more regular architecture making it relatively easier to predict its performance.

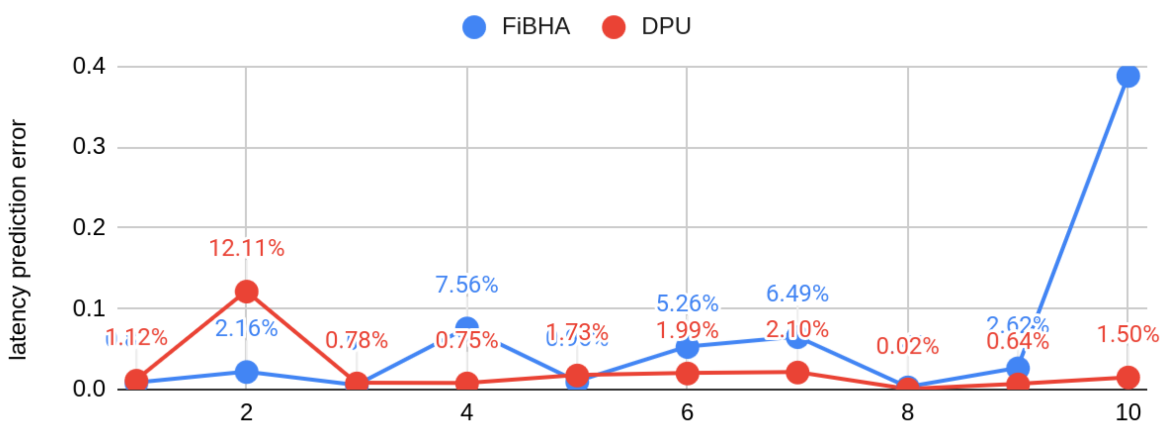


Figure 17: The difference between the predicted and the measured latency on FiBHA of the hardware-aware pruning

Figure 18 shows the achieved speedups and accuracy drop of the pruned models compared to the original model on FiBHA. Up to 2x speedup is achievable with less than 1% loss in accuracy (in the case of model 8). As the figure shows, the same applies to other cases. Figure 19 shows the achieved speedups and accuracy drop of the pruned models compared to the original model on DPU. Up to 2.2x speedup is achievable with less than or equal 1% loss in accuracy (in the case of model 8). The results suggest that finding the best trade-off between accuracy and latency is not trivial. This is because there is no clear correlation between both. For example, model 4 has considerably higher speedup than model 1 and at the same time experiences lower accuracy loss on both accelerators. On FiBHA, the average speedup is 1.5 and the average accuracy loss is 0.9%. On DPU, the average speedup is 1.8 and the average accuracy loss is 1.5%. Generally, both accelerators show similar trends.

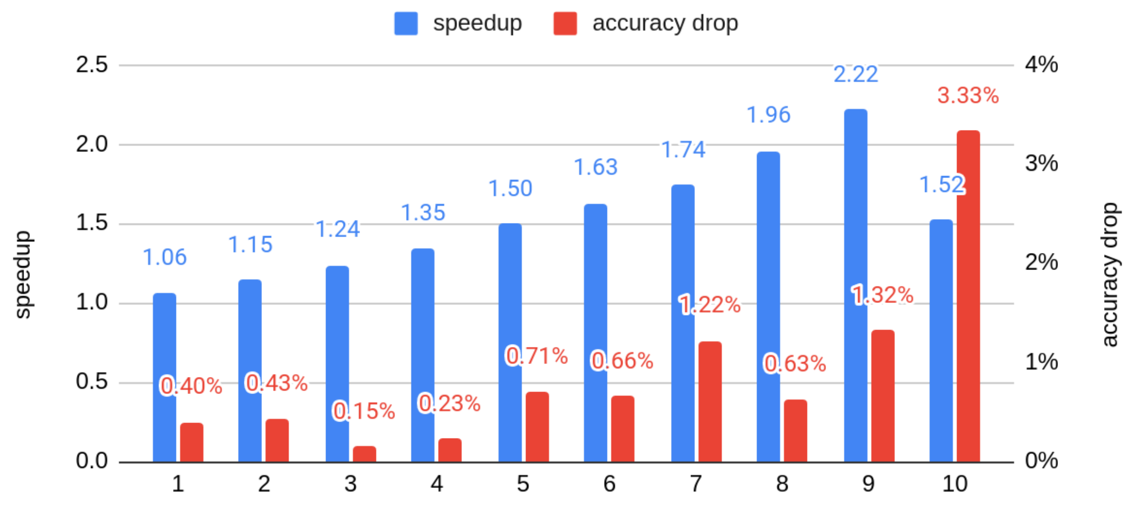


Figure 18: The speedup and accuracy drop of the hardware-aware pruning compared to the original model on FiBHA

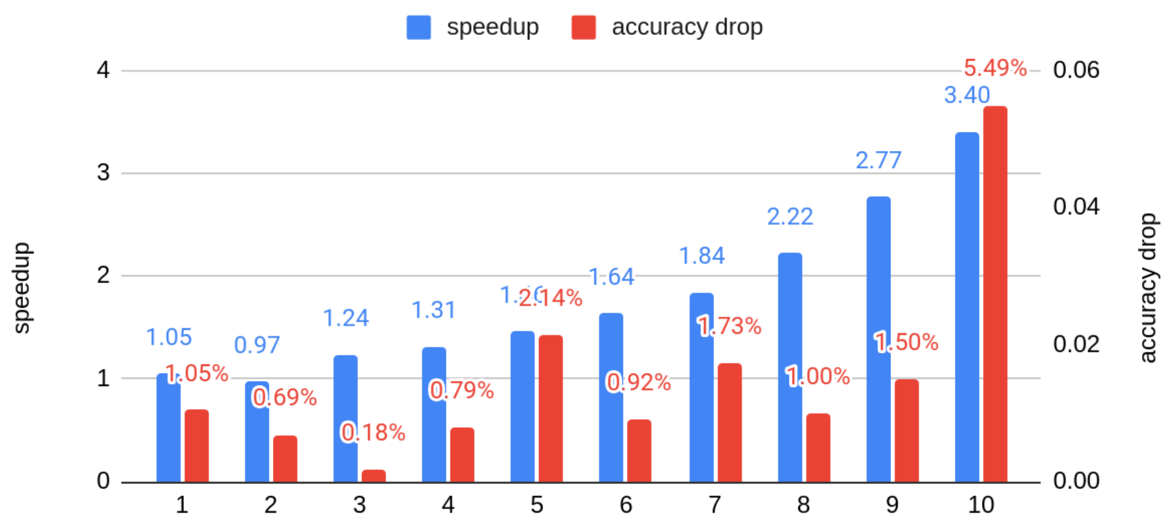


Figure 19: The speedup and accuracy drop of the hardware-aware pruning compared to the original model on DPU

4 Memory architecture and resource allocation for the DL accelerators

DL algorithms pose considerable memory and computing requirements. As such, it is important to analyze and optimize the memory hierarchy. Such an analysis can allow designers to quickly prune the design space to design efficient accelerators. Furthermore, increasing the visibility of this analysis to different phases of the DL algorithm, i.e. layers, not only allows designers to propose efficient data loading methodologies, i.e. RAINBOW's heterogenous plans but allows the runtime to allocate the appropriate amount of resources, i.e. ARADA.

4.1 Memory requirement analysis tool

4.1.1 Introduction

Deep learning (DL), a prominent Machine Learning (ML) algorithm, is increasingly applied to tackle intricate challenges across diverse application domains [30-32]. DL algorithms span the entire compute continuum, from the far edge to the cloud. As DL models grow in size, there is a surge in demands for both computational power and memory. While current systems often incorporate hardware acceleration for computational demands, addressing memory requirements proves more challenging due to space constraints, performance trade-offs, and cost considerations. Additionally, a noticeable shift towards more heterogeneous models necessitates different optimizations for effective data reuse [33]. These factors contribute to the complexity of designing and deploying such applications on target systems.

Existing tools, like SCALE-Sim [34], assist in designing and estimating performance for executing DL applications on dedicated hardware. However, these tools fall short in fully exploring the multi-dimensional space of models and accelerators, often requiring multiple runs to achieve optimization. In response to this multi-dimensional optimization challenge, we introduce RAINBOW, a tool designed to offer pertinent information for hardware-software co-design when executing a DL model on a dedicated hardware accelerator. RAINBOW conducts various analyses, collecting results for essential metrics. These results are then input into an optimizer, which, guided by specific criteria, generates an execution plan that optimally leverages execution characteristics and available resources to achieve its objective. RAINBOW supports distinct optimizers, each tailored to different goals, such as memory utilization. The tool has been presented in [35].

4.1.2 Architecture

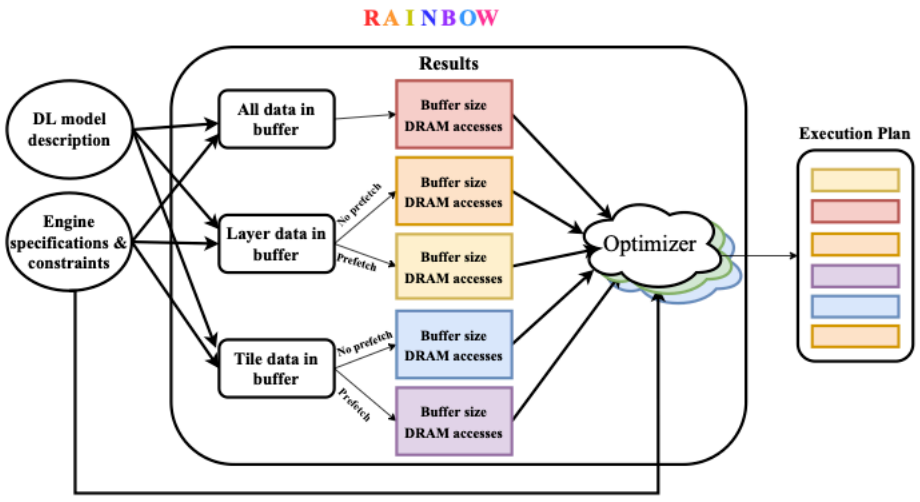

RAINBOW
Results
DL model description
Engine specifications & constraints
All data in buffer
Layer data in buffer
Tile data in buffer
No prefetch
Prefetch
No prefetch
Prefetch
Buffer size DRAM accesses
Buffer size DRAM accesses
Buffer size DRAM accesses
Buffer size DRAM accesses
Buffer size DRAM accesses
Optimizer
Execution Plan

Figure 20: RAINBOW architecture

The RAINBOW tool facilitates hardware-software co-design for on-chip memory in DL accelerators, aiding model designers in determining the deployment methods of a Convolutional Neural Network (CNN) to a dedicated accelerator. The tool's architecture is illustrated in Fig. 20. RAINBOW takes inputs in the form of a CNN description and engine specifications and constraints. Currently, it supports convolution engines and General Matrix Multiplication (GEMM) units, but its adaptable design allows for the potential support of new computational engines. Output files generated by RAINBOW include results from various analyses and execution plans from the optimizers.

For analyses, analytical models were developed to estimate on-chip memory requirements for the CNN and off-chip data transfers across different scenarios. The RAINBOW tool performs five main analyses: (1) All data in buffer; (2) Layer-mode; (3) Tile-mode; (4) Prefetching in Layer mode; and (5) Prefetching in Tile mode. In Layer-mode and Tile-mode, only one layer/tile of feature maps and filters can be stored in on-chip memory at a time. The models assume a layer-by-layer execution, meaning residual connections in some CNNs are serialized.

RAINBOW incorporates optimizers that leverage heterogeneity within the CNN to generate heterogeneous execution plans. These plans better align with optimization goals under design constraints. The optimization goals include achieving optimal data reuse, co-design for multi-precision model quantization, and data reuse with batch execution.

4.1.3 Optimized Execution Plans

4.1.3.1 Homogeneous Plans

Table 3: Homogeneous execution plans

Plan	On-Chip Memory Content
aF/lA: All Filter/Layer Activations	sum(Filter$_{Li}$) + max(Activation$_{Li}$)
aF/tA: All Filter/Tile Activations	sum(Filter$_{Li}$) + max(Activation$_{Ti}$)
aF/tA+p: All Filter/Tile Activation with prefetching	sum(Filter$_{Li}$) + 2*max(Activation$_{Ti}$)
lF/lA: Layer Filter/Layer Activations	max(Filter$_{Li}$ + Activation$_{Li}$)
lF+p/lA: Layer Filter with prefetching/Layer Activation	max(Filter$_{Li}$ + Filter$_{Li+1}$ + Activation$_{Li}$)
lF/tA: Layer Filter/Tile Activations	max(Filter$_{Li}$ + Activation$_{Ti}$)
lF+p/tA: Layer Filter with prefetching/Tile Activaion	max(Filter$_{Li}$ + Filter$_{Li+1}$ + Activation$_{Ti}$)
lF/tA+p: Layer Filter/Tile Activation with prefetching	max(Filter$_{Li}$ + 2*Activation$_{Ti}$)
lF+p/tA+p: Layer Filter with prefetching/Tile Activation with prefetching	max(Filter$_{Li}$ + Filter$_{Li+1}$ + 2*Activation$_{Ti}$)
tF/lA: Tile Filter/Layer Activations	max(Filter$_{Ti}$ + Activation$_{Li}$)
tF+p/lA: Tile Filter with prefetching/Layer Activation	max(2*Filter$_{Ti}$ + Activation$_{Li}$)
tF/tA: Tile Filter/Tile Activations	max(Filter$_{Ti}$ + Activation$_{Ti}$)
tF+p/tA: Tile Filter with Prefetching/Tile Activation	max(2*Filter$_{Ti}$ + Activation$_{Ti}$)
tF/tA+p: Tile Filter/Tile Activation with prefetching	max(Filter$_{Ti}$ + 2*Activation$_{Ti}$)
tF+p/tA+p: Tile Filter with prefetching/Tile Activation with Prefetching	max(2*Filter$_{Ti}$ + 2*Activation$_{Ti}$)

RAINBOW initiates its analysis by examining on-chip memory requirements and DRAM accesses under consideration of homogeneous execution plans. Homogeneous plans involve all layers executing with the same mode, exploring a consistent reuse pattern as described in Section 4.1.2. The various homogeneous plans investigated are detailed in Table 3.

4.2 Sample analysis

In this Section, we will present different analysis results for different software and hardware options that can be available for co-design.

All results presented refer to the analysis for a DPU-like system. These accelerators are named DPU-n where n is the number of operations (addition and multiplication) that the accelerator performs per cycle. So a DPU-n configuration corresponds to an accelerator with n/2 Processing Elements (PEs), in this particular case MAC units, where max n/2 multiply-and-accumulate operations are performed on every cycle. The default configuration for this Section is the DPU-512. Another basic assumption for all the results is that the execution of the models proceeds in a layer-by-layer manner, meaning that each layer is processed completely for a given input and after the whole output is produced then the execution proceeds to the next layer.

4.2.1 Memory requirement

One of the most relevant analysis RAINBOW performs to help accelerator designers is to give an estimate of the required local memory buffer space for filters and activations, for different execution scenarios (as presented in Table 3).

As mentioned above, for this analysis, we use the DPU-512 (256 PEs) as the architecture for the accelerator being modeled. The ML models are assumed to have been quantized to 8-bit integer values. In this analysis we assume to execute homogeneous plans, i.e. all layers will be executed in the same way.

In Figure 21, we show the local memory space needed for five different models (MobileNet, MobileNetV2, Resnet18, Resnet50, and YoloV4) for three different execution scenarios: "*All Filter*" where all filters are stored in local memory, "*Layer Filter + Prefetch*" where the local buffer has space for only the filters for the layer currently executing as well as buffer space for prefetching the filters for the next layer, and "*Layer Filter*" where the local buffer stores only the filters for the layer currently executing. The presented memory values also include the necessary space to store both the input and output activations for the layer being executed.

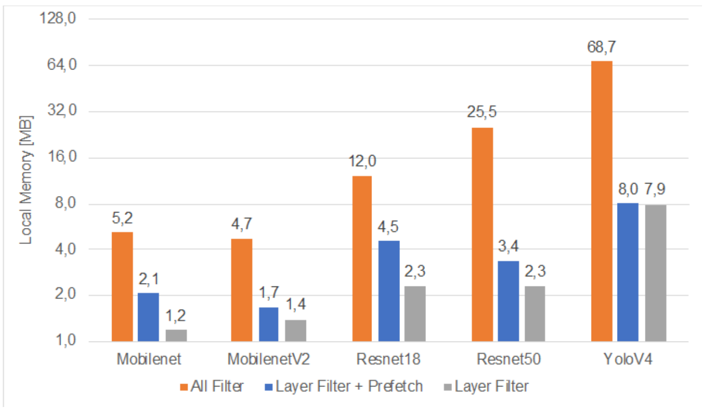


Figure 21: Memory requirements for storing the complete neural networks on the local memory, storing just one layer with prefetching of the weights of the next layer and just storing one layer at a time

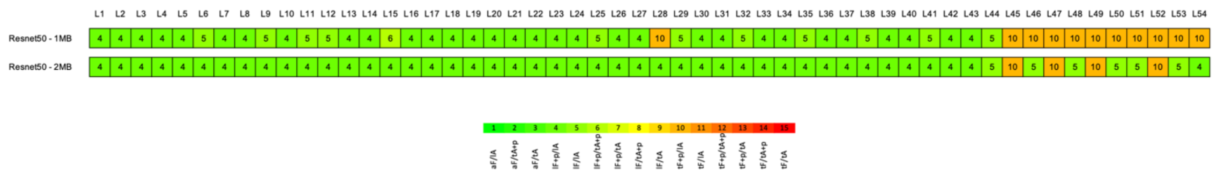
The results in Figure 21 clearly show a large storage requirement difference for the case where we store the complete filters when compared to storing just the layer filters. Note that the memory scale is logarithmic. This difference is larger for the YoloV4 where the complete filters require more than 8x the amount of memory space when compared to storing just the layer filters even when using extra space for prefetching.

So in terms of latency, if the prefetch time can be overlapped with the execution time, the "*Layer Filter + Prefetch*" presents the most interesting design point. Selecting any of the other two options will be determined depending on the local memory space in the budget for the designer. For example, if the designer's budget is 8 MB then it is possible to load all filters for MobileNet and MobileNetV2 but not for the rest. If the memory budget is 4 MB then for Resnet18 it is not possible to save space for prefetching and thus the execution needs to fall back to layer by layer execution without prefetching.

4.2.2 Heterogeneous plans

In this analysis, we explore the benefits of selecting which is the best execution strategy per layer and thus this may result in heterogeneous plans depending on the available memory budget.

We present here results for Resnet50, quantized to 8-bit integer values and the accelerator architecture is as above a DPU-like configured with 256 PEs. We analyze the proposed heterogeneous plans for the cases when we have small available memory space for the local buffer: 1 MB and 2 MB. The case of 4 MB is not relevant since as can be seen in Figure 21, 4 MB is enough to fit the filters for the currently executing layer as well as reserve space for the filters to be prefetched for the next layer.



	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	L16	L17	L18
Resnet50 - 1MB	4	4	4	4	4	5	4	4	5	4	5	5	4	4	6	4	4	4
Resnet50 - 2MB	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

	L19	L20	L21	L22	L23	L24	L25	L26	L27	L28	L29	L30	L31	L32	L33	L34	L35	L36
Resnet50 - 1MB	4	4	4	4	4	4	5	4	4	10	5	4	4	5	4	4	5	4
Resnet50 - 2MB	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

	L37	L38	L39	L40	L41	L42	L43	L44	L45	L46	L47	L48	L49	L50	L51	L52	L53	L54
Resnet50 - 1MB	4	5	4	4	5	4	4	5	10	10	10	10	10	10	10	10	10	10
Resnet50 - 2MB	4	4	4	4	4	4	4	5	10	5	10	5	10	5	5	10	5	4

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
aF/lA	aF/tA+p	aF/tA	lF+p/lA	lF/lA	lF+p/tA+p	lF+p/tA	lF/tA+p	lF/tA	tF+p/lA	tF/lA	tF+p/tA+p	tF+p/tA	tF/tA+p	tF/tA

Figure 22: Heterogeneous execution plans generated by RAINBOW when executing Resnet50 with memory constraint of 1 MB and 2 MB

In Figure 22, we show the heterogeneous plans that are suggested by RAINBOW based on the limited available local memory buffer space. Each different execution strategy corresponds to a different color in the "heat map" where the green represents the strategies that store more values in local memory and thus result in fewer DRAM accesses, and red the opposite. From the results in Figure 22 we can see that for 1 MB space the execution of Resnet50 is using green strategies for most layers, with the exception of the last layers. As the memory space is increased to 2 MB the number of orange strategy layers decreases.

Looking at the number of DRAM accesses for the 1 MB case, the heterogeneous plan is able to reduce the number of DRAM accesses by a factor of 5.7x when compared to the best homogeneous plan for 1 MB. For the 2 MB case, there is no change in the number of accesses from the best homogeneous plan but since the heterogeneous plan includes the use of prefetching in some of the layers, this will result in reduced latency.

4.2.3 Filter and activation quantization

Based on the memory requirement analysis and the available local memory buffer budget constraint, RAINBOW can give a suggestion of what should be the precision bit-width for the filter and activations values for a layer execution strategy (in this case the depicted results are for the execution without the reservation of space for prefetching) for each layer. This information can then be fed back to the query optimizer as to guide a heterogeneous quantization optimization for the particular model and accelerator. In Figure 23 we show the bit-width per layer for the layer data to fit in memory. The starting point is Resnet50 with 32 bits. The solid lines represent the bit width per layer for three different buffer size constraints: 1 MB, 2 MB, and 4 MB. The values presented are not necessarily powers of two as they represent the width that is allowed for the data to fit. This would be acceptable for an FPGA based accelerator with dedicated non-standard computational units but, for example, for the case of a generic GPU, the quantization and operations would have to be rounded to the next lowest acceptable precision. The dashed lines represent some of the standard supported precisions such as 32, 16, and 8 bit.

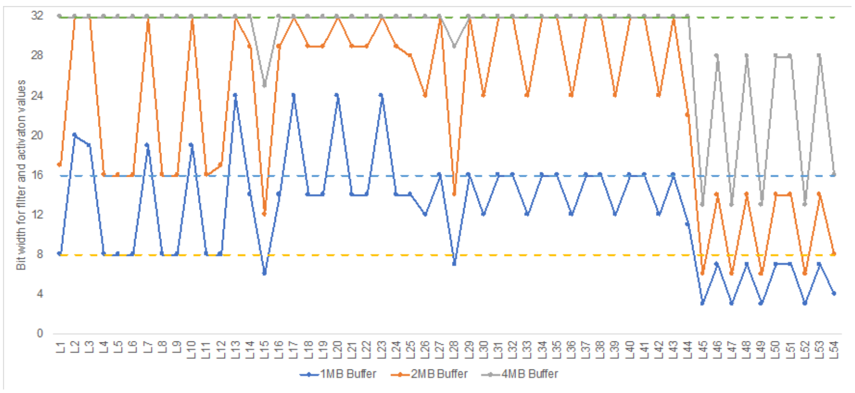


Figure 23: Resnet50 per layer bit-width size for the layer to fit in the local memory of 1, 2 or 4 MB

From the results in Figure 23, it is interesting to observe that the only standard precision that can be used is 8 bit and that requires a buffer of 4 MB. For the case of a buffer with 2 MB, some of the last layers can not fit even when using 8 bits. So for 2 MB, most layers can use 16 bit (except L15 and L28, which would only fit in 8 bit) and the last layers mostly fit in 8 bit with the exception of L45, L47, L49, and L52. This does not necessarily mean that we need to support a precision smaller than 8 bit but instead that the execution of those layers when using 8 bit will not be able to fit completely in the buffer so the execution will be done using a tiling strategy.

4.2.4 Number of processing elements

In this analysis we show the impact on the execution time of the number of PEs in the DPU configuration that is adopted. For this analysis we assume a large enough buffer size so that the memory does not affect the results.

In Figure 24 we present the compute time for the execution of all layers of the model for five different models. The y-axis represents the time in number of cycles and the scale is logarithmic.

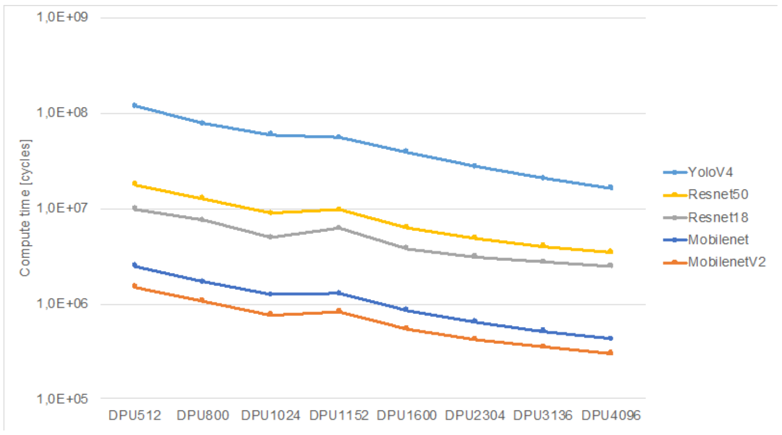


Figure 24: Scaling of execution time (cycles) of different neural networks for various DPU configurations

As expected the computing time decreases as the number of PEs increases. Notice just a small “unexpected” trend for the values for the DPU1152 configuration. The DPU1152 does not perform better than the DPU1024, even though it can perform more operations per cycle. The difference is because of the first layer of the models. The DPU1024 has a pixel parallelism (PP) of 8 and Input Channel Parallelism (ICP) and Output Channel Parallelism (OCP) of 8 as well. On the other hand, the DPU1152 has a PP of 4 and ICP and OCP of 12. The impact on the results comes from the fact that the first layer has only three channels, which leads to higher parallelism for the DPU1024, as it has higher PP.

4.2.5 Pruned model

In this section we show the analysis of the impact of pruning on memory requirements and computing time. In particular we show the analysis for some of the pruned models from the WP6 optimizations as a part of the co-design effort (Section 3.5), in particular using uniform and non-uniform pruning.

In Figure 25 we show the memory requirement for the MobileNetV2 model, for 50% uniform and non-uniform pruned models, normalized to the original non-pruned model.

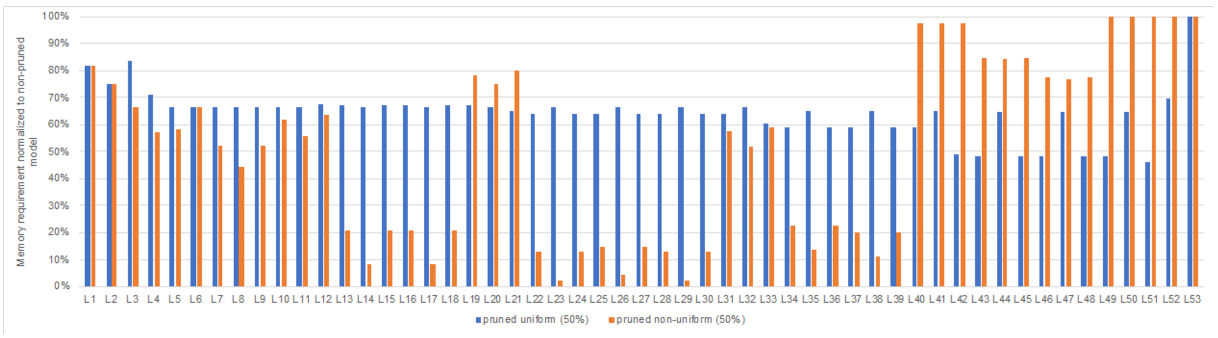
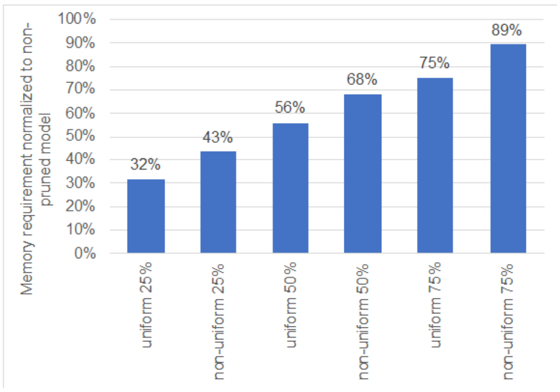


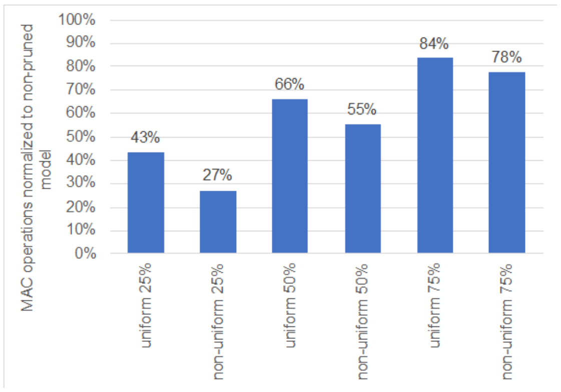
Figure 25: MobileNetV2 per layer memory requirements for uniform and non-uniform pruning normalized to non-uniform model

From Figure 25, it is possible to observe that the uniform pruned model follows almost the same trend as the original model while the non-uniform differs quite a lot from layer to layer. This offers some potential for better utilization of the internal buffer space and also in terms of the design of pushing pruning hints to the optimization so that we can manage to fit certain more demanding layers into the available buffer space.

In Figure 26, we show the memory requirement for the layer strategy execution and the number of MAC operations for the MobileNetV2 model with uniform and non-uniform pruning of 25%, 50%, and 75%, normalized to the non-pruned model.



(a) Memory requirements of the different pruning strategies for storing in local memory one layer at a time, normalized to the non-pruned model



(b) Number of MAC operations for the different pruning strategies normalized to the non-pruned model

Figure 26

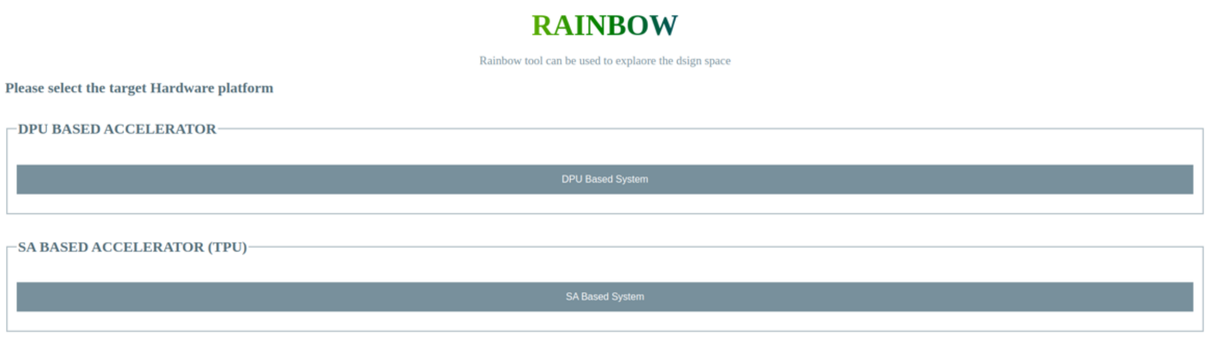
The results show that, as expected, as we go from more aggressive (25%) to less aggressive (75%) pruning, memory requirements and MAC operations increase. The interesting observation though is that the non-uniform models always require more memory but less MAC operations. If we observe the per layer memory requirements in Figure 25, we see that non-uniform pruning requires larger memory size in order to store the layer parameters, especially for the first and last layers of the model. However, the middle layers have a really small number of parameters which leads to severely underutilized memory, and a really small amount of MAC operations, reducing the total number of MAC operations for the model.

4.3 Memory tool GUI interface

In order to demonstrate the capabilities of RAINBOW, we have prepared a web application that allows, in a very intuitive way, to test different configurations of DL accelerators. This application interfaces with the RAINBOW repository using the DJANGO framework.

4.3.1 Interface and results

The interface of the RAINBOW demo is depicted in Figure 27. The designers can now choose a DPU or SA-based system architecture.


RAINBOW

Rainbow tool can be used to explaore the dsign space

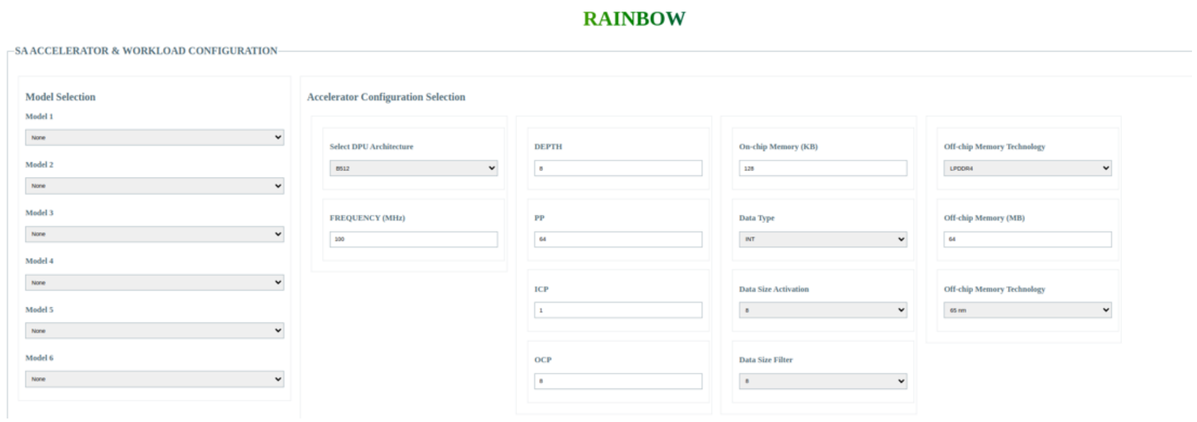
Please select the target Hardware platform

DPU BASED ACCELERATOR

SA BASED ACCELERATOR (TPU)

Figure 27: Start Interface for RAINBOW Demo web application

The available options for DPU-based analysis are shown in Figure 28. Users can select multiple models and choose various DPU configurations from Xilinx, data type, off-chip memory technology, and on-chip BRAM size. The analysis will be done based on the data type specified by the user. It is interesting to note that we support different data types for weights & OFM/IFM. Moreover, users can specify the advanced filter and IFM loading option. RAINBOW also supports a heterogeneous plan.


RAINBOW

SA ACCELERATOR & WORKLOAD CONFIGURATION

Model Selection

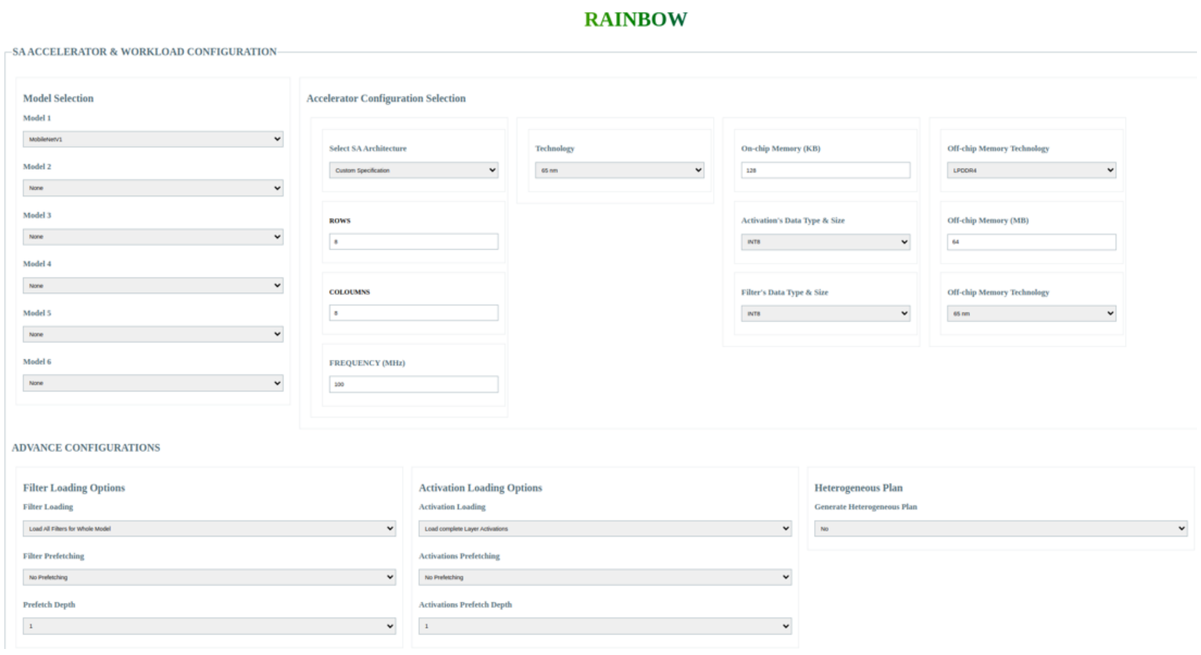
Model 1: None
Model 2: None
Model 3: None
Model 4: None
Model 5: None
Model 6: None

Accelerator Configuration Selection

Select DPU Architecture: B512
FREQUENCY (MHz): 100
DEPTH: 8
PP: 64
ICP: 1
OCP: 8
On-chip Memory (KB): 128
Data Type: INT
Data Size Activation: 8
Data Size Filter: 8
Off-chip Memory Technology: LPDDR4
Off-chip Memory (MB): 64
Off-chip Memory Technology: 65 nm

Figure 28: RAINBOW Demo Interface for DPU-based Analysis

The available options for SA-based analysis are shown in Figure 29. Again the user can select the parameters concerning the SA-based architecture. Here, a few popular SA-based architectures such as Edge TPU, and TPU v1-v5 are present. Users can choose custom SA architecture as well. Moreover, as in the case of DPU, there is an option to select off-chip memory technology and on-chip buffer sizes. Users also specify the clock frequency. The energy calculations are based on the CACTI tool.



RAINBOW

SA ACCELERATOR & WORKLOAD CONFIGURATION

Model Selection

Model 1: MobileNetV1

Model 2: None

Model 3: None

Model 4: None

Model 5: None

Model 6: None

Accelerator Configuration Selection

Select SA Architecture: Custom Specification

ROWS: 8

COLOUMNS: 8

FREQUENCY (MHz): 100

Technology: 65 nm

On-chip Memory (KB): 128

Activation's Data Type & Size: INT8

Filter's Data Type & Size: INT8

Off-chip Memory Technology: LPDDR4

Off-chip Memory (MB): 64

Off-chip Memory Technology: 65 nm

ADVANCE CONFIGURATIONS

Filter Loading Options

Filter Loading: Load All Filters for Whole Model

Filter Prefetching: No Prefetching

Prefetch Depth: 1

Activation Loading Options

Activation Loading: Load complete Layer Activations

Activations Prefetching: No Prefetching

Activations Prefetch Depth: 1

Heterogeneous Plan

Generate Heterogeneous Plan: No

Figure 29: RAINBOW Demo interface for SA-based Analysis

For all the architecture types, RAINBOW demo creates graphs for latency, energy, required buffer sizes, and off-chip memory access as shown in Figure 30.

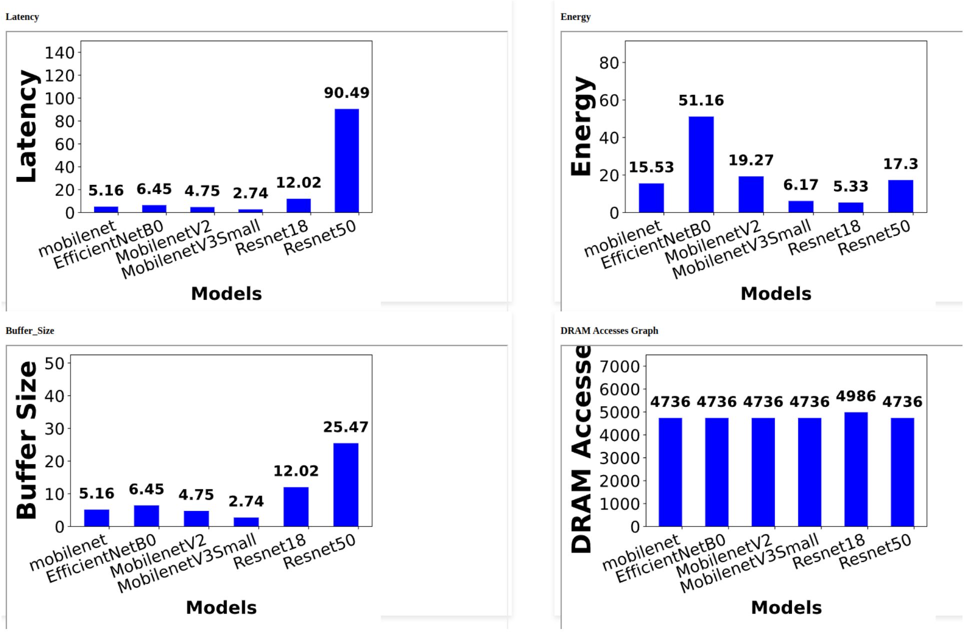


Figure 30: RAINBOW Demo Results

4.4 Adaptive resource allocation for DL accelerators

The efficient execution of a DL model requires matching compute and memory speed, as this ensures zero stalls in both compute and memory subsystems. However, it is not possible with static resource allocation for two reasons. First, the different phases of the DL algorithm require different computations or memory speeds and thus require different resource allocations. Second, new DL algorithms are continuously developed and executed on existing architectures and thus dynamic adaptation of the resources to the hardware is required. Both these factors highlight the need for resource allocation for different phases of DL applications.

The scenario of different compute and memory requirements is depicted in Figures 31, 32, and 33. Here we plot the roofline model for various layers of EfficientNetB0 for three different configurations of an accelerator with an architecture similar to the Google TPU: (1) 0.4 TOPS compute and DDR-4000 memory (Figure 31); (2) 0.4 TOPS compute and HBM-1 memory (Figure 32); and (3) 4 TOPS compute and HBM-1 memory (Figure 33). The DDR4-4000 offers a memory bandwidth of 29GB/sec while the HBM-1 offers a memory bandwidth of 122 GB/sec. There are several important insights here. First, the different phases have different compute and memory bandwidth requirements. In this context, we can note that some phases or layers are compute-bound, and some are memory-bound. Furthermore, the extent of memory and computing boundness is different for various layers.

Lastly, we can observe that when accelerator architecture is changed, i.e., compute performance changes from 0.4 TOPS to 4 TOPS or memory bandwidth from 29 GB/sec to 122 GB/sec, the layers change from compute to memory bound and vice versa.

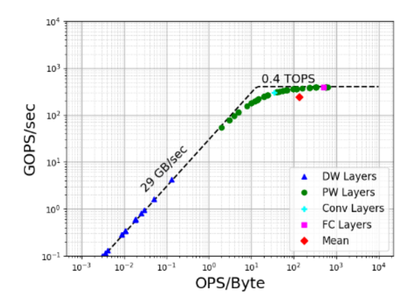


Figure 31: Per layer roofline plot for EfficientNetB0 for 0.4 TOPS compute performance and DDR4-4000 (i.e., 29 GB/sec) memory.

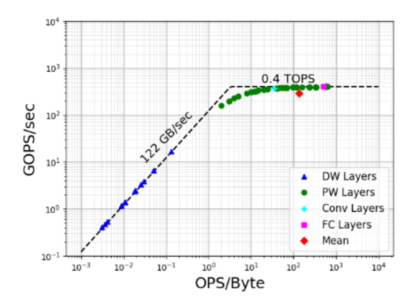


Figure 32: Per layer roofline plot for EfficientNetB0 for 0.4 TOPS compute - HBM-1 (i.e.122 GB/sec) memory.

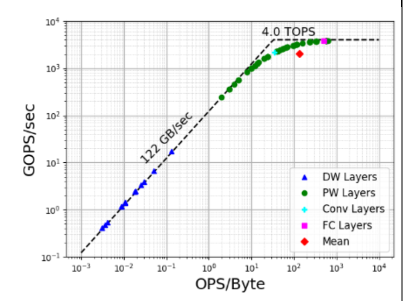


Figure 33: Per layer roofline plot for EfficientNetB0 for 4 TOPS compute - HBM-1 (i.e.122 GB/sec) memory.

In short, these varying compute and memory requirements across different phases of a model and for different accelerators require runtime resource allocation. Therefore, ARADA proposes to analyze the DL workloads and propose resource allocations for each phase or layer. Figure 34 depicts the architecture of ARADA, which takes the DL model and hardware specifications as input and generates a resource allocation schedule, i.e., V-F for the computing engine and memory bandwidth allocation for each layer within the model. These allocations are collected in a table and passed to the runtime system. At runtime, the resource manager takes these decisions with the help of the system monitor. Further details on the implementation can be found in [18].

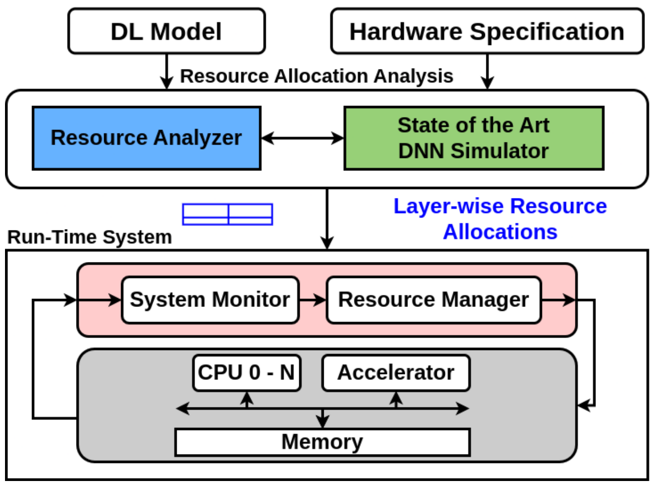
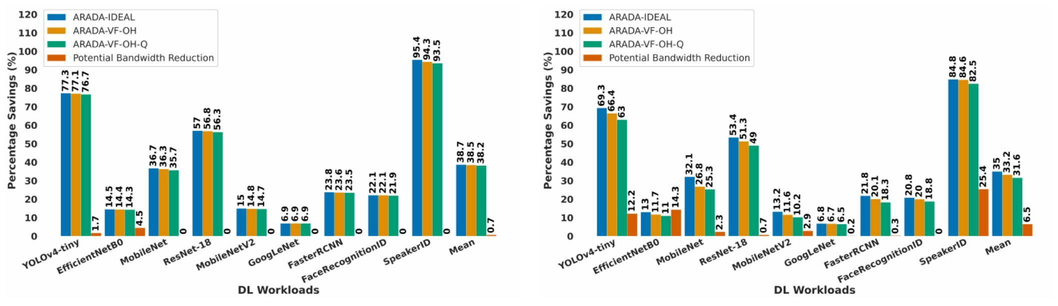


Figure 34: ARADA System Architecture

This scheme was evaluated for two architectures on two ends of the computing spectrum: Edge TPU with 4 TOPS and HPC TPU with 123 TOPS compute performance. These accelerators are coupled with several memory types providing various bandwidths resulting in a range of architectures. ARADA is compared with the race to idle (RTI) and oracle, i.e., ARADA-IDEAL. ARADA-IDEAL is a perfect scheme used to demonstrate the upper limit of savings on offer. Two variants of ARADA are analyzed. First, *ADARA-VF-OH* deducts the overheads for DVFS scaling. Second, ADARA-VF-OH-Q deducts DVFS overheads and considers a quantized voltage-frequency scaling. ScaleSim simulator is used to establish the phases that are compute- and memory-bound. This analysis is fed to an in-house resource analyzer to generate the schedule.

The results for Edge TPU are shown in Figure 35, where the x-axis shows the models, while the percentage savings for energy and bandwidth are shown on the y-axis. As a first observation, as we increase the BW starting from LPDDR4 in Figure 35.a to DDR5-4800 in Figure 35.b, the potential energy savings decrease, and potential BW reduction rises as expected. Moreover, each model has its distinct behavior, and BW changes have a different effect. However, the degree of change due to memory technology is workload-dependent. Thus, estimating and applying resource allocation to each workload per layer level is essential. Another critical point is the effect of DVFS overhead and V-F quantization.

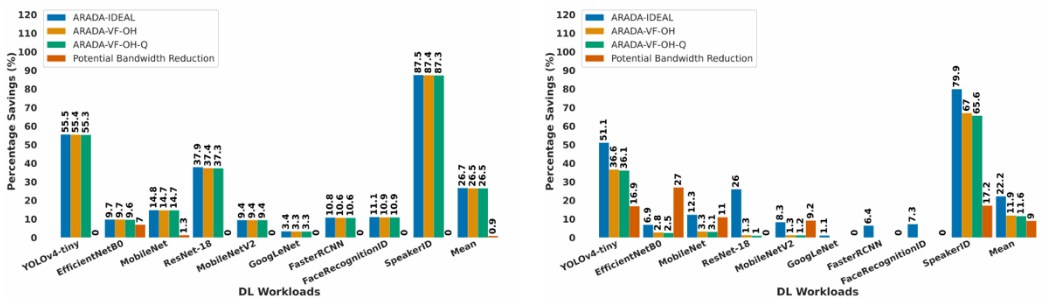
On the other hand, as bandwidth increases, the potential energy savings decrease because of shorter stall times for the compute engine. This also results in further reduction in the case of ARADA-DVFS-OH and ADARA-VF-OH-Q schemes, as when the stall time is less than DVFS overhead, reduction in V-F is not feasible. Stall times depend on the total execution time of a layer or, in other words, the layer size. Therefore, the effect is diverse for various workloads. For example, the reduction in energy savings from ARADA-IDEAL to ARADA-VF-OH-Q is different for YOLOv4-tiny and GoogLeNet for LPPDDR4 and DDR5 4800 cases.



(a) LPDDR4 (b) DDR5 4800

Figure 35: Energy Savings for Edge TPU with 4 MB on-chip buffer and various off-chip memory types

Figure 36 depicts the results for HPC TPU, where the x-axis and y-axis show the workloads and the percentage savings for energy and bandwidth, respectively. Again, we deliver results for various memory types, i.e., DDR4 4400 in Figure 36.a and HBM2 in Figure 36.b. Furthermore, as we increase the BW, more layers shift from memory-bound to compute-bound regions. Models behave differently because they have a unique mixture of compute and memory-bounded layers. Furthermore, the extent to which a particular layer is compute- or memory-bound differs, resulting in individual behavior. Accounting for DVFS overhead and quantization again reduces the energy savings. However, as can be seen in the case of Figure 36.b, this reduction is significant. The reason is that HBM2 has a very high bandwidth, significantly reducing stall times. Resource allocation is not applied if stall times are less than DVFS overheads. Therefore, there is a significant difference between the DDR4-4400 and HBM2. Thus, it is essential to analyze each workload in connection with the underlying architecture and to allocate resources accordingly at runtime.



(a) DDR4 4400 (b) HBM2

Figure 36: Energy Savings for various DL workloads for HPC TPU with 8MB on-chip buffer for different off-chip memory types

We present a per-layer analysis to investigate the detailed, intricate behavior further, showing the potential reduction in the TOPS and BW. In this context, we offer results for MobileNet for executing on HPC TPU with DDR5-4800 and HBM2 in Figure 37.a and Figure 37.b, respectively. There are several interesting observations here. First, as discussed earlier, most DL application layers are compute-bound or memory-bound. Nevertheless, some layers still need both computation and memory, indicating they are at an efficient execution point for the given configuration. This is the case, for example, for layers 10-24 for HPC TPU with DDR5-4800, as shown in Figure 37.a. As the memory BW increases in HPC TPU with HBM2, some of these layers shift to compute-bound regions. For example, layers 13, 15, and

17 are moved to the compute-bound region. The second important observation is that the various layers have different degrees of compute-bounded or memory-bounded behavior, resulting in different potential reductions in energy and bandwidth. In this context, it is essential to note that this phenomenon is affected by the hardware platform, as can be seen by comparing the two figures. For example, layer 13 has approximately 25% BW reduction in the case of HBM2 and none in the case of DDR5-4800.

Similarly, looking at layer 5, one can observe a TOPS reduction of 90% for DDR5-4800 and 60% for HBM2. Switching the off-chip memory from DDR5-4800 to HBM2 allows data to be moved faster, reducing the memory-bounded behavior of the layer. In other words, the stall time waiting for data is decreased, as expected.

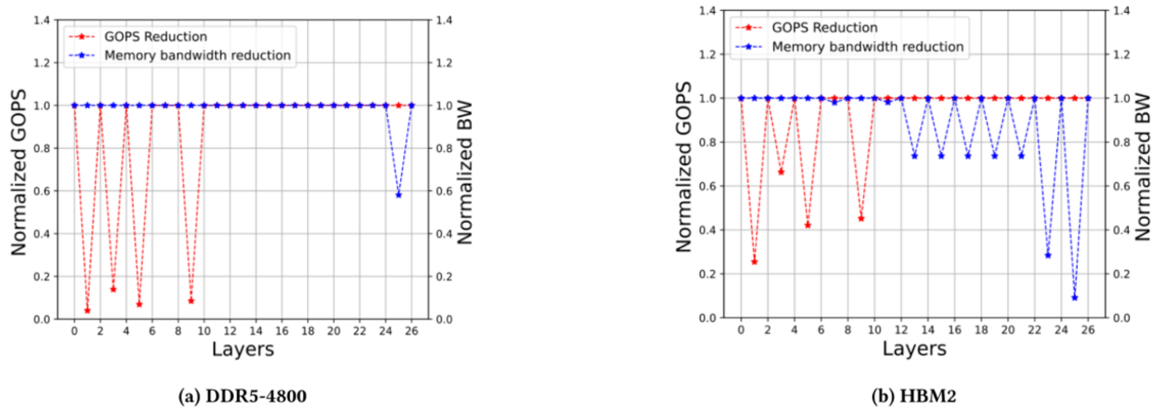


Figure 37: Layer-wise compute and bandwidth reductions for MobileNet executing on 256 × 256 SA with 8 MB on-chip buffer

5 Conclusion

In this document, we have summarized the efforts by the VEDLIoT partners on the development of efficient Deep Learning accelerators. We propose different accelerators which are integrated into the VEDLIoT platforms developed in WP4. The design of the accelerators is also done in close collaboration with the inputs and optimizations proposed by the DL toolkit developed in WP6.

In terms of accelerators we start by presenting different off-the-shelf accelerators that were considered for our project. Then we present FPGA-based accelerators which not only are customized statically to the target application but also are able to dynamically change their configuration at runtime depending on changes in the dynamic requirements and different operational modes (e.g. idle versus active and day versus night object recognition).

Most of our proposed accelerators follow the co-design approach as to obtain more efficient implementations. STANN and FiBHA are two examples of such accelerators. This co-design has been exploited fully with a iteration loop between the VEDLIoT toolkit model optimizations and the reconfigurable accelerator designs.

Since the number of parameters in modern models increases considerably, the correct design and management of the memory hierarchy are also key aspects of the accelerator design. In this document we presented the efforts related to the development of a tool that helps guide the design exploration of different options in the memory hierarchy. In addition, we have also presented a methodology to adapt the number of resources for the different hardware requirements at the different phases of the execution of a model.

Overall, the accelerators proposed within the context of this project have advanced the state-of-the-art with solutions that are providing solutions dedicated to the target applications and are able to exploit different techniques to achieve high efficiency for the execution of the DL models.

6 References

[1] Qararyah, F., Azhar, M. W., & Trancoso, P. (2022, November). Fibha: fixed budget hybrid CNN accelerator. In 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (pp. 180-190). IEEE.

[2] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. Leong, andP. Y. Cheung, “Redundancy-reduced mobilenet acceleration on recon-figurable logic for imagenet classification,” in International Symposium on Applied Reconfigurable Computing. Springer, 2018, pp. 16–28.

[3] S. Yan, Z. Liu, Y. Wang, C. Zeng, Q. Liu, B. Cheng, and R. C.Cheung, “An fpga-based mobilenet accelerator considering network structure characteristics,” in 2021 31st International Conference onField-Programmable Logic and Applications (FPL). IEEE, 2021, pp.17–23.

[4] L. Bai, Y. Zhao, and X. Huang, “A cnn accelerator on fpga using depthwise separable convolution,” IEEE Transactions on Circuits andSystems II: Express Briefs, vol. 65, no. 10, pp. 1415–1419, 2018.

[5] J. Liao, L. Cai, Y. Xu, and M. He, “Design of accelerator for mo-bilenet convolutional neural network based on fpga,” in 2019 IEEE 4thAdvanced Information Technology, Electronic and Automation ControlConference (IAEAC), vol. 1. IEEE, 2019, pp. 1392–1396.

[6] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan,“A high-performance cnn processor based on fpga for mobilenets,” in2019 29th International Conference on Field Programmable Logic andApplications (FPL). IEEE, 2019, pp. 136–143.

[7] J. Gao, Y. Qian, Y. Hu, X. Fan, W.-S. Luk, W. Cao, and L. Wang, “Leta:A lightweight exchangeable-track accelerator for efficientnet based on fpga,” in 2021 International Conference on Field-Programmable Tech-nology (ICFPT). IEEE, 2021, pp. 1–9.

[8] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, “An fpga-based cnnaccelerator integrating depthwise separable convolution,” Electronics,vol. 8, no. 3, p. 281, 2019.

[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 Proceedings of the 2017 ACM/SIGDA interna-tional symposium on field-programmable gate arrays, 2017, pp. 65–74.

[10] 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 Transactions on Reconfigurable Technology and Systems (TRETS),vol. 11, no. 3, pp. 1–23, 2018.

[11] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, “A high perfor-mance fpga-based accelerator for large-scale convolutional neural net-works,” in 2016 26th International Conference on Field ProgrammableLogic and Applications (FPL). IEEE, 2016, pp. 1–9.

[12] S. I. Venieris and C.-S. Bouganis, “fpgaconvnet: A framework for map-ping convolutional neural networks on fpgas,” in 2016 IEEE 24th AnnualInternational Symposium on Field-Programmable Custom ComputingMachines (FCCM). IEEE, 2016, pp. 40–47.

[13] 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,” arXiv preprint arXiv:1704.04861, 2017.

[14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 4510–4520.

[15] M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for con-volutional neural networks,” in International conference on machine learning. PMLR, 2019, pp. 6105–6114.

[16] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally,and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size,” arXiv preprint arXiv:1602.07360,2016.[12] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” arXiv preprint arXiv:1812.00332,2018.

[17] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. Leong, and P. Y. Cheung, “Redundancy-reduced mobilenet acceleration on recon- figurable logic for imagenet classification,” in International Symposium on Applied Reconfigurable Computing. Springer, 2018, pp. 16–28.

[18] Muhammad Waqar Azhar, Stavroula Zouzoula, and Pedro Trancoso. 2023. ARADA: Adaptive Resource Allocation for Improving Energy Efficiency in Deep Learning Accelerators. In Proceedings of the 20th ACM International Conference on Computing Frontiers (CF '23).

[19] AMD Xilinx. 2023. DPUCZDX8G for Zynq UltraScale+ MPSoCs Product Guide (PG338). https://docs.xilinx.com/r/en-US/pg338-dpu Accessed: 2023-08-25.

[20] AMD Xilinx. 2023. DPUCVDX8G for Versal ACAPs Product Guide (PG389). https://docs.xilinx.com/r/1.1-English/pg389-dpucvdx8g Accessed: 2023-08-25.

[21] AMD Xilinx. 2023. DPUCADF8H for Convolutional Neural Networks Product Guide (PG400). https://docs.xilinx.com/r/en-US/pg400-dpucadf8h Accessed: 2023-08-25.

[22] VEDLIoT, Deliverable D3.3 Evaluation of the DL accelerator designs, 2022.

[23] VEDLIoT, Deliverable D4.6 Final report on FPGA infrastructure, RISC-V system and accelerators, 2023.

[24] VEDLIoT, Deliverable D7.5 Final report on use case development, optimisation, benchmarking and evaluation

[25] Marc Rothmann and Mario Porrmann. 2023. STANN - Synthesis Templates for Artificial Neural Network Inference and Training. In 17th International Work-Conference on Artificial Neural Networks (IWANN2023). Springer International Publishing.

[26] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers.. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’17). ACM, 65–74.

[27] Javier Duarte et al. 2019. Fast Inference of Deep Neural Networks for Real-Time Particle Physics Applications. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Seaside, CA, USA) (FPGA ’19). Association for Computing Machinery, New York, NY, USA, 305. https://doi.org/10.1145/3289602.3293986

[28] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. 2017. Low-memory GEMM-based convolution algorithms for deep neural networks. ArXiv abs/1709.03395 (2017).

[29] VEDLIoT, Deliverable D 7.8 Report on Open Call results, 2023

[30] 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,” arXiv preprint arXiv:1704.04861, 2017.

[31] J. Kim, J. K. Lee, and K. M. Lee, “Accurate image super-resolution using very deep convolutional networks,” in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 1646–1654.

[32] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, “Convolutional neural networks for speech recognition,” IEEE/ACM Transactions on audio, speech, and language processing, vol. 22, no. 10, pp. 1533–1545, 2014.

[33] A. Boroumand, S. Ghose, B. Akin, R. Narayanaswami, G. F. Oliveira, X. Ma, E. Shiu, and O. Mutlu, “Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks,” in 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, 2021, pp. 159–172.

[34] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, “A systematic methodology for characterizing scalability of dnn accelerators using scale-sim,” in 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2020, pp. 58–68.

[35] S. Zouzoula, M. W. Azhar and P. Trancoso, "RAINBOW: Multi-Dimensional Hardware-Software Co-Design for DL Accelerator On-Chip Memory," 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Raleigh, NC, USA, 2023, pp. 352-354, doi: 10.1109/ISPASS57527.2023.00050.