CNNParted: An Open Source Framework for Efficient Convolutional Neural Network Inference Partitioning in Embedded Systems

Fabian Kreß, Vladimir Sidorenko, Patrick Schmidt, Julian Hoefer, Tim Hotfilter, Iris Walter, Tanja Harbaum, Jürgen Becker

[a]*Karlsruhe Institute of Technology, Karlsruhe, Germany*

Abstract

Applications such as autonomous driving or assistive robotics heavily rely on the usage of Deep Neural Networks. In particular, Convolutional Neural Networks (CNNs) provide precise and reliable results in image processing tasks like camera-based object detection or semantic segmentation. However, to achieve even better results, CNNs are becoming more and more complex. Deploying these networks in distributed embedded systems thereby imposes new challenges, due to additional constraints regarding performance and energy consumption in the near-sensor compute platforms, i.e. the sensor nodes. Processing all data in the central node, however, is disadvantageous since raw data of camera consumes large bandwidth and running CNN inference of multiple tasks requires certain performance. Moreover, sending raw data over the interconnect is not advisable for privacy reasons. Hence, offloading CNN workload to the sensor nodes in the system can lead to reduced traffic on the link and a higher level of data security.

However, due to the limited hardware-resources on the sensor nodes, partitioning CNNs has to be done carefully to meet overall latency requirements and energy constraints. Therefore, we present CNNParted, an open-source framework for efficient, hardware-aware CNN inference partitioning targeting embedded AI applications. It automatically searches for potential partitioning points in the CNN to find a beneficial workload distribution between sensor nodes and a central edge node. Thereby, CNNParted not only analyzes the CNN architecture but also takes hardware components, such as dedicated hardware accelerators and memories, into consideration to evaluate inference partitioning regarding latency and energy consumption.

Exemplary, we apply CNNParted to three commonly used feed forward CNNs in embedded systems. Thereby, the framework first searches for several potential partitioning points and then evaluates the latter regarding inference latency and energy consumption. Based on the results, beneficial partitioning points can be identified depending on the system constraints. Using the framework, we are able to find and evaluate 10 potential partitioning points for FCN ResNet-50, 13 partitioning points for GoogLeNet, and 8 partitioning points for SqueezeNet V1.1 within 520 s, 330 s, and 140 s, respectively, on an AMD EPYC 7702P running 8 concurrent threads. For GoogLeNet, we determine two partitioning points that provide a good trade-off between required bandwidth, latency and energy consumption. We also provide insights into further interesting findings that can be derived from the evaluation results.

Keywords: Convolutional Neural Networks, Embedded Systems, Hardware Accelerator, Simulation Framework, Hardware/Software Co-Design

1. Introduction

In the last decade, Deep Neural Networks (DNNs) have been the center of attention in research focused on image processing. This is due to the high level of accuracy achieved by these networks and the possibility to deploy them in many different use cases, e.g. autonomous driving, intelligent prosthetics and assistive robotics [1]. However, due to the increasing computational complexity of these networks [2], their use in embedded hardware presents new challenges.

It is no longer feasible to deploy such networks on general-purpose CPUs or GPUs, since these platforms are not able to provide sufficient inference latency and throughput. Instead, dedicated accelerators as well as hardware/software-co design are necessary techniques to allow the efficient usage of these Convolutional Neural Network (CNN) models. Through these means, it is possible to increase performance and energy efficiency by more than 100x and 1000x, respectively [3]. Still, the compute power necessary to execute these algorithms continues to increase.

Several emerging applications rely on a combination of multiple sensors, such as cameras, lidars and radars, instead of single static images in order to perceive their environment. In assistive robotics, the focus lies on personalization and real-time capabilities to achieve user acceptance. The robot utilizes cameras and other sensors in order to recognize people and to perceive

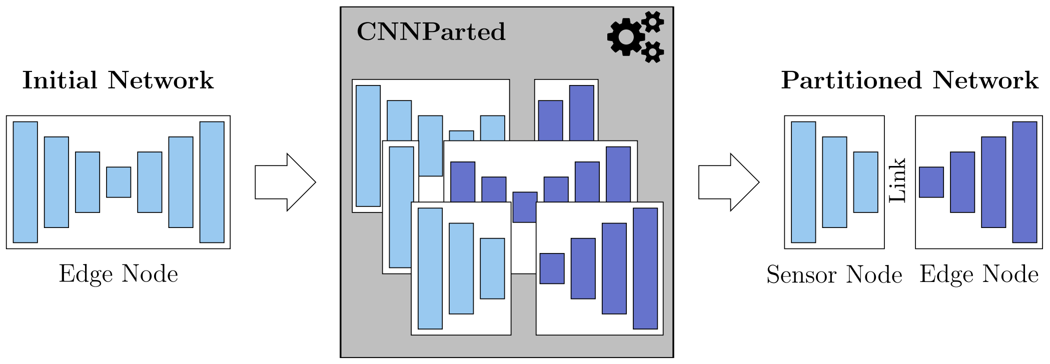

Figure 1: CNNParted automatically evaluates various partitioning points of a CNN model on multiple nodes. Both, sensor and edge node are embedded platforms with the former being constrained in energy consumption and area. The link layer transmits intermediate results and can be inserted in between any network layer. As a result, CNNParted outputs metrics for multiple potential partitioning points that help the designer efficiently distribute the workload.

its environment. Real-time recognition is facilitated through local execution in the robot by dedicated accelerators [4] instead of remote computations. This also enhances privacy of the data.

Additionally, autonomous driving is another use-case that benefits from CNNs. Data from different sensors is evaluated by a DNN in order to perform object detection and semantic segmentation. The evaluated sensor data can then be used for driver-assistance systems such as lane assistants, face recognition [5] and also fully autonomous vehicles. The different sensor nodes are all connected to a centralized Electronic Control Units (ECUs) by an on-board network. It is desirable to reduce the amount of traffic on the network, hence performing parts of the DNN computation directly in the sensor node is advantageous. This approach has the added benefit of increased data security while simultaneously keeping the communication latency low, since reconstructing raw input data from intermediate feature maps requires deep knowledge of the CNN model and its parameters.

To deploy CNN-based applications in embedded and distributed systems, we present CNNParted[1], an open source framework which searches for beneficial partitioning points in a feed forward CNN and evaluates them based on a given system architecture. As shown in Figure 1, the framework evaluates

[1]https://github.com/itiv-kit/cnn-parted

several configurations, where the first partition of CNN layers is executed on a sensor node, while the second partition is processed on a central edge node. Apart from evaluating latency and energy consumption of both compute platforms, CNNParted also considers the link in between, which is used to transfer the output feature map of the last CNN layer processed in the sensor node to the edge node. This paper is an extended version of our work presented in [6], where we studied the partitioning of CNNs in general and presented a toolchain to evaluate inference partitioning for several feed forward CNNs. The extended contributions in this paper are as follows:

- We present CNNParted, an open source simulation framework for evaluating energy and latency of CNN partitioning in embedded systems
- We show the application of CNNParted to obtain latency and energy consumption for different CNN inference partitioning
- We evaluate our tool by automatically identifying potential partitioning points of multiple, typical feed forward CNNs, namely FCN-ResNet50, GoogLeNet, and SqueezeNet
- We give insights into beneficial partitioning of these CNNs for different system setups, evaluating also the impact of various link configurations

The remainder of this paper is divided into four sections. In Section 2, we analyze the current state of the art to highlight the relevance of CNNParted. Section 3 gives a detailed overview of the concept and implementation of the framework. In Section 4, we evaluate the performance of CNNParted for three different CNNs. Finally, Section 5 concludes this paper and provides an outlook on the next steps.

2. Related Work

Since DNN inference imposes heavy computational and energy requirements it is not feasible to execute the full network on the sensor nodes [7]. It follows that these operations must be moved to other, more powerful devices. To ease the computational burden, several accelerator architectures specific for co-deployment in sensor nodes have been proposed. An example of this is RedEye, an analog accelerator that is connected directly to the pixel array of a camera [8]. It is equipped with dedicated functional units to perform the operations most commonly found in CNNs, i.e. computations, max pooling, and is also capable of performing quantization operations. Partitioning of a GoogLeNet between RedEye and a second computational node, a Jetson TK1 GPU, showed an improved latency while simultaneously reducing the overall power consumption slightly.

Besides of designing specialized accelerators for embedded platforms to meet latency and energy constraints, research has also been carried out considering DNN inference partitioning over multiple nodes. However, these mainly take off-the-shelf platforms like GPUs or Field-Programmable Gate Array (FPGA) based hardware IP modules for DNN inference partitioning into account and mostly target only datacenter applications [9, 10, 11, 12, 13]. Moreover, approaches such as Distributed Deep Neural Networks (DDNNs) proposed by Teerapittayanon et al. consider DNN inference partitioning already during training, showing reduced communication overhead by maintaining high accuracy [14].

Ghasemi et al. propose a framework for energy-efficient partitioning of a DNN between a user device and a cloud server [15]. Their goal is to minimize the energy consumed by the user device during computation as well as transmission while executing a network of DNNs. The problem is formulated as a weighted flow graph and the optimal partitioning point is calculated by a min-cut algorithm. However, their approach relies on previous profiling of the power requirements of the user device. For each new accelerator, the above step must be repeated to determine the energy consumption of the hardware used.

An edge-host inference partitioning approach has been presented by Ko et al. [16]. They designed and synthesized an inference engine containing 144 16-bit MAC units, an on-chip buffer and a JPEG encoder and decoder which allows storing weights in compressed format. They were able to prove that DNN partitioning can be beneficial regarding throughput and energy-efficiency. However, they do not provide a hardware-aware design space exploration for DNN partitioning.

Similarly, the Deep Compressive Offloading framework, presented by Yao et al., aims for reducing link latency by adding lightweight encoder and decoder in between the partitioned neural network [17]. Even though the results show very small accuracy loss, the authors only evaluate latency and energy consumption for off-the-shelf platforms in the sensor node, i.e. two different Android phones.

Hu et al. propose the usage of autoencoder-based compression to optimize throughput and accuracy of a pipelined CNN inference [18]. Similar to the approach presented in this paper, the first inference stage is placed in transmitting node, i.e. the sensor, while the second is performed by the receiver. Thereby, to reduce communication overhead between both platforms, an Autoencoder is inserted. However, their approach focuses on computation load-balance but does not consider power consumption or the accelerator embedded near the sensor. This would be helpful when identifying optimal partitioning points of the CNN.

Zhao et al. present DeepThings, an open source framework to split CNN inference across multiple edge nodes leading to lower local memory requirements as well as drastic improvements on performance [19]. The network is automatically split into multiple, independent tiles which are then distributed to the various nodes while considering their individual compute resource constraints. Still, the authors only consider methods to optimize scheduling while ignoring the hardware implementation of the accelerator. It follows that this method only improves latency and throughput by dynamic load balancing while considerations of energy across the different nodes are neglected.

A co-exploration method for hardware and neural architecture co-design targeting real-time applications is introduced by Yang et al. [20]. To achieve this, the network is split across multiple FPGAs connected by a network-on-chip. A feedback loop is used in the exploration step to improve accuracy and hardware efficiency. However, their approach again does not consider energy needed by the hardware and can therefore not be used to optimize energy efficiency of embedded Artificial Intelligence (AI) scenarios.

An approach to partition multiple CNNs across a network of IoT devices is proposed by Disabato et al. [21]. The network of devices is modelled as a graph where vertices are only connected if they are in communication range. The goal is to minimize the total latency of the data processing. To this end, the individual latencies of computation and communication are calculated and the optimization is formulated as a binary linear program. The solution is calculated by a solver. Similar to previously discussed approaches, the authors disregard energy requirements which makes this approach not suitable when trying to find an energy efficient partitioning.

To the best of our knowledge, there is a lack of tools that can find a beneficial CNN inference partitioning point considering the deployed hardware architecture and the link between compute platforms. Hence, in this paper, we propose CNNParted, a hardware- and link-aware framework for determining efficient workload distribution of a CNN in embedded AI applications.

3. CNNParted

As already mentioned, several embedded applications like autonomous driving or assistive robotics rely on multiple sensors to achieve a good perception of the environment. With the rising number of sensors deployed in such systems, the amount of captured data increases and, hence, sending raw data to an other node in the network for processing becomes disadvantageous. Consequently, recent embedded systems consist of distributed compute platforms enabling data preprocessing close to the sensor. As a result, the amount of data to be transferred is significantly reduced, yet computationally intensive applications such as semantic segmentation still have to be performed in a central computing node.

In applications such as autonomous driving, there are multiple tasks running in parallel requiring a very powerful compute node to meet the latency constraints. Overcoming this limitation can be achieved by offloading even more workload from the central compute platform to the sensor nodes including e.g. CNN inference. Consequently, this approach not only results in lower link utilization but also takes the limited compute resources in the central platform into account. Overall, partially offloading CNN workload to the sensor node helps to achieve lower link usage and higher system performance.

However, especially sensor nodes are constrained in energy consumption as well as in available area on the chip. In order to allow for operation on limited energy and area, highly specialized hardware accelerators have to be deployed in the sensor node. This leads to an increased complexity of the hardware/software co-design, since partitioning of CNN inference also has to take hardware requirements into account to achieve high energy efficiency and low link utilization. In the following, we therefore present CNNParted, an open source framework for automated design space exploration of inference partitioning in distributed compute platforms.

The aim of the framework is to find a good trade-off for CNN inference partitioning in terms of latency, energy consumption, required link bandwidth and workload distribution. Providing a full system model is thereby not required, since usage of constraints, e.g. maximum available bandwidth, allows to take the impact of other nodes on the link into account. Hence, modelling of the actual system can be simplified by solely considering a near-sensor node (*Sensor Node*) which is connected to a central compute platform (*Edge Node*) over a specific link. An overview of CNNParted is provided in Figure 2.

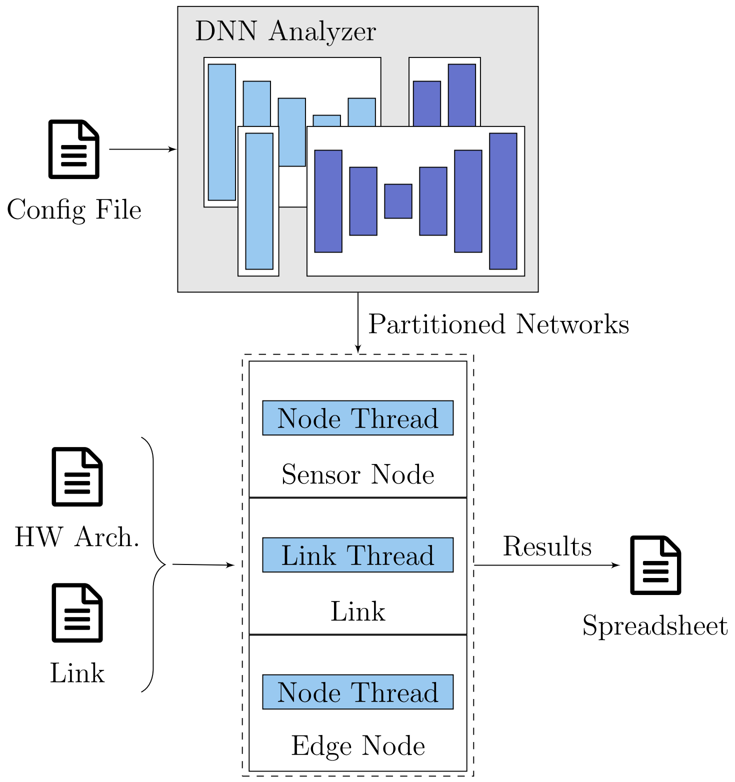

Figure 2: Overview of CNNParted. The framework takes a configuration file as input containing the CNN architecture and the node parameters. First, the DNN Analyzer determines several partitioning points of the CNN, then the hardware-aware evaluation of these points is performed. Therefore, CNNParted takes the architecture description and a link configuration as input and outputs results for energy consumption and latency.

CNNParted is designed to be flexible and extensible allowing for the integration of a broad range of system simulation and hardware/software co-design tools into the workflow. Its goal is to find beneficial partitioning points in a feed forward CNN, however, it can also be used for other DNNs by replacing CNN-related tools and adapting the search algorithm accordingly. As input, CNNParted takes a single configuration file containing the following information:

- The CNN to be analyzed as well as the input size.
- The sensor and edge node configuration, including the CNN layer name to be run on the specified hardware accelerator at a certain frequency

and the tool settings for evaluating the energy consumption and latency. In case a real platform should be used for evaluation, only the device, the number of threads and the number of evaluation runs for determining the median latency have to be defined.

- The link configuration including, e.g. the data bit width, as well as the specific communication protocol parameters.

- A constraint for limiting the maximum output feature map size to be sent over the link, i.e. the maximum available transmission bandwidth, to account for other nodes in the system architecture. The list of constraints can be extended depending on user requirements, e.g. by specifying a maximum layer depth for searching beneficial partitioning points or a maximum number of parameters to account for the limited memory capacity in the sensor node.

After reading the configuration file, CNNParted first analyzes the CNN architecture and determines potential partitioning points. The corresponding module is called DNN Analyzer. Subsequently, the resulting set of partitioned models is evaluated regarding energy consumption and latency for each part of the system, i.e. sensor node, link, and edge node. Thereby, to reduce the overall run time of CNNParted, the evaluation of each system module runs in a separate thread. Finally, the evaluator collects the simulation results of the preceding step and generates estimates of the energy consumption and latency for each partitioned model. As output, CNNParted provides the data in spreadsheet format to the system designer for determining the best partitioning point regarding the system constraints. We will provide some insights into the interpretation of these results in Section 4.

3.1. DNN Analyzer

Manually setting possible partitioning points in CNNs takes a huge amount of time, especially for large model architectures consisting of more than a hundred layers. In addition, partitioning can only be done if the architecture is known to the tool which is not always the case. Since CNNParted is meant to be used for any feed-forward CNN, a methodology is required to automatically determine potential partitioning points in unknown neural architectures. CNNParted therefore contains a module to analyze the given neural network called DNN Analyzer, which is based on PyTorch.

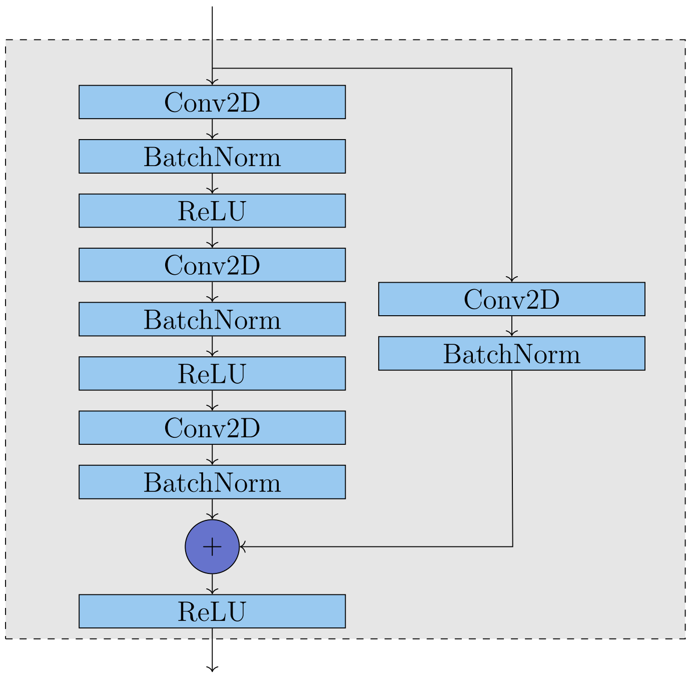

Figure 3: Example of a residual layer containing skip connection taken from FCN ResNet architecture [22]. Partitioning within such blocks is usually disadvantageous, since this requires higher link bandwidth.

To retrieve basic information of the given CNN, a forward pass including forward hooks is performed on the model using the torchinfo module. It outputs a list of the configuration of all layers and their sublayers, if any exists, which is used to further analyze the architecture of the CNN. For flattened feed forward models containing no sublayers determining potential partitioning points is trivial. However, partitioning of neural networks is not advisable for each individual layer or layer type, respectively. State-of-the-art networks often use *skip connections* as shown in Figure 3 to prevent vanishing gradients during training. Skip connections are usually grouped into building blocks. However, partitioning the network within these building blocks would introduce a larger amount of data to transfer and a greater memory footprint. Hence, potential partitioning points are usually placed in between the aforementioned building blocks and in layers without skip connections.

Consequently, the DNN Analyzer has to find these building blocks based on the architecture description. Therefore, the CNN is converted into a graph representation to allow for depth-first search of the layer levels. The recursive function implementing this search, shown in Algorithm 1, adds each leaf layer on the current level directly to the list of potential partitioning points.

Algorithm 1: Depth-first search to identify potential partitioning points in the given CNN architecture.

```
function GetPartPoints
Input  : CNN architecture, root module
Output: potential partitioning points

found ← True

if root module is leaf layer then
    insert root module into partPointList
else
    found ← FindPartitions(root module)                // call Algorithm 2
end

if found is True then
    for each neighbor in CNN architecture do
        GetPartPoints(CNN architecture, neighbor)      // recursive call
    end
end
```

Open source machine learning frameworks such as PyTorch offer multi-level DNN architectures based on special layer types, e.g. ModuleDict or Sequential [23], to implement residual layers or skip connections, it is also necessary to iterate through their child layers. Hence, the DNN Analyzer evaluates the forward pass of all modules containing sublayers to recognize such connections in the CNN. The implementation of this is presented in Algorithm 2.

Thereby, the submodules of the given CNN layer are each appended to a list of layers building a feed forward neural network architecture without any branches, i.e. DNN Analyzer reorganizes the model into a sequential block. This is then evaluated as well as the original module using a randomly initialized tensor. If both models return the same output tensor, all direct sublayers can be added to the list of potential partitioning points, since no skip connections have been found. Subsequently, Algorithm 1 continues its search by evaluating the neighbors of the given module. Otherwise, if the models return different tensors after evaluation, the depth-first search

Algorithm 2: Evaluation of modules to recognize partitioning points between submodules.

```
function FindPartitions
Input  : CNN module

seq ← new List

for each m in module.children do
    append m to seq
end

rand_tensor ← random tensor
res1 ← evaluate(seq, rand_tensor)
res2 ← evaluate(module, rand_tensor)

if res1 and res2 are equal then
    if module in partPointList then
        replace module in partPointList by submodules
    else
        insert all submodules in partPointList
    end
    return True
end

return False
```

stops at this point and moves on to the next neighbor looking for potential partitioning points until all branches have been explored.

Finally, the DNN Analyzer analyzes the output feature map size of each partitioning point and filters out all CNN layers that do not meet the corresponding constraint. The resulting list contains all potential partitioning points of the CNN and can be accessed by other classes in the CNNParted framework for evaluating energy consumption and latency of the sensor and edge node, as well as for the link in between these platforms.

3.2. Node Evaluation

CNNParted assumes an embedded system setup which consists of a data processing node close to the sensor, the sensor node, and a central compute platform, the edge node. In contrast to the development of server hardware architectures, which mainly focus on low latency and high throughput, several additional constraints must be considered in the design of embedded systems. Especially due to the limited energy consumption in the sensor node, an extensive hardware/software co-design is particularly important to meet performance requirements of the system. Therefore, achieving high energy efficiency for CNN inference partitioning requires a hardware-aware evaluation of the system.

Usually, Convolution (CONV) layers contribute by far the majority of the computational operations to most CNNs. For example, as shown by Guo et al, in VGG-11, they account for more than 98% of the operations performed [24]. Therefore, the evaluation of the embedded platform within CNNParted focuses on CONV layers to estimate energy consumption and latency as well as the required bandwidth on the link between sensor and edge node. Consequently, this methodology offers a first step towards reduced simulation time, which is very important in the context of hardware/software co-design.

CNNParted provides a common API which allows to adapt different tools to the workflow for evaluating important metrics, i.e. latency and energy consumption of the potential CNN inference partitions. Consequently, tools other than the two presented in the following can be easily integrated into CNNParted if needed.

3.2.1. Model-based Node Evaluation

In embedded hardware platforms, specialized accelerators are often deployed to meet performance requirements of the corresponding application. Thereby, these architectures are optimized towards high energy efficiency, high throughput, and low latency. In recent years, many different CNN inference hardware accelerators have been proposed [25, 26, 27, 28, 29], few of them being open source as well [30, 31]. Each of them has been designed for either a single application or a range of use cases offering more flexibility. However, since each of the architectures has its advantages and drawbacks, such as power and area consumption, performance, and flexibility, it is not trivial to determine a suitable design for the CNN workload given the system constraints.

CNNParted requires a highly flexible and rapid simulation framework to evaluate important metrics of these CNN hardware accelerators for various workloads including search for optimal mapping on the architecture. In general, several frameworks have been proposed offering cycle-accurate simulation and evaluation of Application-Specific Integrated Circuits (ASICs) [32, 33, 34] for DNN inference. Even though these come with high accuracy of the estimated metrics, the major drawback is the enormous simulation time for analyzing each cycle individually. Hence, this is not a viable approach to evaluate different layers of a given CNN rapidly.

Consequently, CNNParted focuses on giving first estimates to support the system design process. Trading accuracy for significantly reduced simulation time thereby allows to evaluate a broader range of hardware accelerators for CNN inference partitioning. There are several of such higher-level simulation frameworks available, most of which are also available as open source [35, 36, 37, 38, 39, 40]. However, most of these tools lack flexibility because they are tied to a specific hardware accelerator architecture or offer only a few adjustable parameters, resulting in limited support for ASICs.

In our proposed framework, we use Timeloop [41] together with Accelergy [42] since this provides a fully flexible hardware design space as well as fine-grained analytical models to produce good estimates for the most important metrics. Based on a given model of the hardware accelerator, Timeloop first searches for an optimal inference mapping of a single CONV layer and then evaluates the corresponding latency. Therefore, CNNParted first extracts all relevant layers as well as the corresponding input shape and then launches Timeloop for analysis. As an output of the mapping, the tool also gives detailed statistics about each individual module of the accelerator allowing also for estimating the energy consumption of the ASIC. However, since Timeloop itself only provides very basic component libraries, other tools are required to provide more reliable estimates for each CNN layer. Accelergy is an open source tool for evaluating energy consumption based on action counts of each instantiated module. It provides primitive component libraries and calls estimation plugins for generating energy consumption estimates for each primitive. Within CNNParted, we integrate the CACTI [43] plugin for evaluating energy consumption and latency of the memories inside the accelerator. Thereby, the tool is able to generate good estimates for these metrics since it takes the architecture as well as the technology parameters of the memory into account. For modelling the power consumption and latency of the logic components integrated in the accelerator, CNNParted uses the Aladdin [44] plugin. This accelerator simulator allows for evaluation of the design without the need for an RTL description based on the construction of a Dynamic Data Dependence Graph (DDDG). Overall, integrating Timeloop and Accelergy hence allows for fair comparison of different ASICs for each CNN layer in a reasonable amount of time.

3.2.2. Measurement-based Node Evaluation

As already mentioned, since many hardware accelerators remain non open source, accurate models are not always available. In addition, accounting for embedded systems containing only a CPU or a GPU is not possible using simulation tools such as Timeloop. Hence, CNNParted provides a module called Generic Node which performs latency measurements on real platforms instead of performing model-based simulations. We therefore apply PyTorch Benchmark in our framework [23].

Apart from the possibility of running multiple threads to reduce overall run time, it can ensure good performance metrics through warm-up run and CUDA synchronization at startup. Further, Python is available for many platforms and operating systems ensuring the execution of CNNParted on a broad range of embedded systems. This approach consequently allows designers to deploy either high-performance computing platforms, e.g. NVIDIA Jetson TX-2, or general-purpose embedded CPU platforms, such as a Raspberry Pi, depending on the workload and use case. Nevertheless, real measurements on Commercial-Off-the-Shelf (COTS) platforms usually have the disadvantage of significant statistical variances that can occur due to several other tasks being executed in parallel on these operating systems. To overcome this, CNNParted therefore determines the median value of 1000 measurements to achieve more reliable inference latency estimations.

However, accounting for the power consumption of the node is less accurate in this configuration because COTS platforms typically implement general-purpose components that introduce noticeable overhead. In addition to peripherals, the operating system and tasks running in parallel have a significant impact on overall system power consumption. CNNParted therefore neglects this metric for the Generic Node, as it assumes constant power consumption.

3.3. Ethernet Link Model

With the modular structure of the framework, various models of the partition interconnection link can be implemented and easily integrated into the toolflow. Conceptually, this is the link that carries feature maps between two neighbouring layers of the CNN, where the partitioning takes place.

In order to estimate the impact of the link on the overall performance of the modelled CNN, link power consumption and transmission latency are taken into account. In the current implementation, copper-based Ethernet is assumed as the connection link, although development of other link models, e.g. Bluetooth, is considered in near future. Data is assumed to be transferred over the link in a deterministic fashion, i.e. with a certain amount of data being transmitted over equal intervals of time.

Power estimation for standard Ethernet is based on typical power consumption values of Physical layer (PHY) devices and latency is calculated based on queuing time with the defined line rate and cable propagation delay [6]. Thus, transmission latency model depends on four user-defined parameters: overall amount of data produced by the source CNN layer, maximum Ethernet packet size, link bandwidth and cable length.

Nevertheless, with improvement of energy efficiency being one of the primary purposes of the CNNParted framework, the model is enhanced with a generalized Energy-Efficient Ethernet (EEE) model based on Equation 1 to compute mean power consumption $E[P_{EEE}]$ [45]. Not being restricted to specific energy-saving strategies, this model allows to explore various approaches and improve power consumption of the link on the conceptual level.

$$P_{EEE} = P \cdot (1 - (1 - \sigma_{off})(1 - \rho))\frac{T_{off}}{T_{off} + T_S + T_W} \tag{1}$$

Here, mean baseline power consumption E_P is that of the legacy Ethernet. Link utilisation ρ is dependent on the use case and must be less than one for correct functioning of the model. σ_{off} represents the ratio between power consumption in normal and Low-Power Idle (LPI) operation modes of the PHY device and estimated to be 0.1 [45, 46]. An important input parameter of the model, $E[T_{off}]$ is the mean duration of LPI mode and is defined by an energy-saving strategy and must be specified explicitly. Two more parameters, T_S and T_W are the active-to-sleep and sleep-to-active transition times respectively. The current model uses the minimum T_S and T_W values as specified in the corresponding standard [47].

As the deterministic traffic model is assumed, latency evaluation does not differ significantly from the legacy Ethernet. As long as the link is never overloaded, it is sufficient to only add the link wake-up time T_W to the legacy latency model. To ensure that the link bandwidth is never exceeded at a specific feature map transmission rate F_{FM} and line rate B, the following condition must be satisfied:

$$\frac{N_{data}}{B} + T_W + T_s \leq \frac{B}{F_{FM}} \tag{2}$$

The EEE-enabled latency model is described with Equation 3, where T_{wire}

being the cable propagation delay.

$$T_{EEE} = T_W + \frac{N_{data}}{B} + T_{wire} \tag{3}$$

Based on these equations, CNNParted can estimate the latency and energy consumption of the link and its components for each potential partitioning point of the CNN. Overall, this approach enables a bandwidth-aware evaluation of CNN inference partitioning during design time.

4. Evaluation

In the following, we show the application of CNNParted for inference partitioning of typical feed forward CNNs. The evaluation is executed on two different systems, especially providing a good comparison regarding framework runtime. One is based on a 64-core AMD EPYC 7702P, the second system is built of an octa-core Intel Xeon W-2145 and an NVIDIA RTX A6000 which is designed and optimized for visual computing. Both systems are running Rocky Linux and are executing CNNParted using the same tools, i.e. Python, PyTorch, Timeloop and Accelergy including CACTI and Aladdin plug-ins.

For the model-based node evaluation, we use always the same Timeloop configuration to allow for a fair comparison. The tool runs eight threads in parallel and targets a delay- and energy-optimal mapping of each CONV layer of the given CNN for a specific hardware accelerator architecture. Thereby, we make use of the linear-pruned search algorithm, which uses pruning techniques to improve efficiency of the linear search. It is terminated as soon as the victory condition is fulfilled, meaning 100 consecutive valid but suboptimal mappings have been determined. Overall, this configuration offers good results while achieving low simulation time.

Since we want to consider different types of hardware accelerators in our evaluation, we use two distinct architectures in the following. The more powerful Simba-like architecture [48] is clocked at 500 MHz and provides good performance for large CNNs. It is based on a weight-stationary dataflow and uses 1024 Processing Elements (PEs). Optimized towards low energy consumption, we choose to also evaluate an Eyeriss-like architecture [49] which is clocked at 200 MHz and offers suitable latency and throughput for smaller CNNs. In contrast to the Simba-like architecture, it applies row-stationary dataflow and consists of 256 PEs for processing CNN layers.

4.1. Workloads

For the evaluation, we select three commonly used CNNs: Fully Convolutional Network (FCN) ResNet-50 [22] for semantic segmentation as well as GoogLeNet [50] and SqueezeNet V1.1 [51] designed for application in image classification tasks. When loading into CNNParted, the framework adds an *Identity* layer in the beginning of each CNN to consider evaluation of executing all layers on the edge node.

ResNets can be configured at different levels of complexity. FCN ResNet usually uses ResNet-50 or ResNet-101 configuration as backbone, achieving 91.4% or 91.7% pixel accuracy, respectively. In general, the network topology consists of configuration independent head and tail layers and four *Bottleneck* blocks. The latter contains a certain amount of residual blocks, which is determined by the configuration. In contrast, the network head is built of a large convolution and max pool to reduce the dimensions, and a batch normalization layer whereas the tail consists of two CONV layers with batch normalization. As already mentioned, since partitioning in between residual layers is not beneficial, the submodules of the *Bottleneck* layers are not considered as potential partitioning points in CNNParted. Hence, this results in 25 partitioning points for FCN ResNet-50 determined by the DNN Analyzer. Furthermore, if the maximum output shape at the split point is restricted to a feature map with 450,000 elements, 10 possible configurations remain.

In a similar fashion, potential partitioning points are determined for GoogLeNet. It consists of multiple *Inception* modules which provide multiple paths between the layers, with different convolution filter sizes. Hence, partitioning within these modules is not beneficial. Apart from the *Inception* modules, the network architecture includes simple CONV, pooling and dense layers. For GoogLeNet, the DNN Analyzer determines 19 partitioning points. If the maximum output shape at the split point is restricted to a feature map with 200,000 elements, CNNParted finds 13 possible partitioning points.

Finally, SqueezeNet V1.1 has been designed towards a small memory footprint containing a very low amount of parameters. The basic building block of SqueezeNet is the *Fire* module which performs efficient feature extraction implementing concurrent 1x1 and 3x3 CONV layers. With a CONV layer at the beginning and end as well as pooling operations before each *Fire* module, SqueezeNet V1.1 offers 17 partitioning points. Assuming a maximum output shape of 150,000 elements in the feature map, this results in 8 potential split points.

4.2. CNNParted Runtime Analysis

The integration of CNNParted into hardware/software co-design workflows is only feasible if the evaluation of multiple partitioning points is not overly time-consuming. By providing good estimates for latency and energy consumption, fast simulation enables a broad exploration of the design space to determine suitable system architectures. Therefore, the framework may add only a marginal overhead to the integrated tools for analyzing CNN layers and potential partitioning points. In the following, we analyze the runtime of both parts, determining beneficial partitioning points of a given CNN and evaluation of these in terms of energy consumption and latency. For this, we ran each simulation at least 10 time to ensure certain reliability of the results. Since the link evaluation is based on a mathematical model, estimating relevant metrics takes about a millisecond for all configurations and is therefore neglected.

Table 1: Median runtime and standard deviation of DNN Analyzer on a CPU and a GPU to analyze the model architecture for three different CNNs out of 30 runs.

Architecture	**AMD EPYC 7702P**	**NVIDIA RTX A6000**
FCN ResNet-50	2081.84 ms ($\sigma = 32.78\ ms$)	108.88 ms ($\sigma = 0.90\ ms$)
GoogLeNet	372.22 ms ($\sigma = 14.09\ ms$)	100.46 ms ($\sigma = 0.76\ ms$)
SqueezeNet V1.1	246.10 ms ($\sigma = 0.74\ ms$)	68.79 ms ($\sigma = 1.01\ ms$)

The median runtime of the DNN Analyzer, the first module of the proposed framework, for the three CNN workloads is shown in Table 1. As expected, since actual execution of the forward pass is performed to find branches in the CNN architecture, the GPU outperforms the CPU. Nevertheless, even the CPU-based system offers good performance for larger models like the FCN ResNet-50 and only takes about 2.1 seconds to search for potential partitioning points. Consequently, the results prove that the chosen approach to search for possible split points works reasonably fast.

Following the workflow of CNNParted, after analyzing the CNN architecture, the nodes are evaluated in terms of energy consumption and latency for each partitioning point. Table 2 shows the simulation runtime of Timeloop for the two different hardware accelerators on both evaluation systems. Obviously, since the Simba-like accelerator is more complex and therefore offers a larger mapspace, the exploration takes more time than for the Eyeriss-like accelerator.

Table 2: Median simulation time and standard deviation of two CPUs to evaluate all CONV layers using Timeloop for three different CNN architectures and two hardware accelerators out of 10 runs.

Architecture	**Eyeriss-like Accelerator**		**Simba-like Accelerator**	
	Xeon W-2145	EPYC 7702P	Xeon W-2145	EPYC 7702P
FCN ResNet-50	307.59 s ($\sigma = 2.62\ s$)	169.64 s ($\sigma = 0.60\ s$)	857.36 s ($\sigma = 7.32\ s$)	519.04 s ($\sigma = 1.10\ s$)
GoogLeNet	302.34 s ($\sigma = 35.81\ s$)	161.03 s ($\sigma = 0.49\ s$)	563.02 s ($\sigma = 8.36\ s$)	329.67 s ($\sigma = 0.98\ s$)
SqueezeNet V1.1	143.87 s ($\sigma = 1.61\ s$)	70.28 s ($\sigma = 0.22\ s$)	229.89 s ($\sigma = 6.99\ s$)	138.70 s ($\sigma = 0.28\ s$)

The results are based on running Timeloop using 8 concurrent threads for exploration. Consequently, slight runtime differences are observed between the Intel Xeon W-2145 and the AMD EPYC 7702P for both accelerators. However, we also ran simulations where 32 threads were executed in parallel. Although Timeloop is able to find better mappings in terms of power consumption and latency by using more threads, the runtime can be almost twice as high. For FCN ResNet-50, the exploration based on Simba-like architecture takes about 770 seconds, while for GoogLeNet we observe a runtime of 630 seconds. In contrast, we measured a slight improvement in simulation runtime with SqueezeNet V1.1. This is due to the fact that Timeloop uses these additional threads to explore even more possible mappings, which can result in the victory condition being fulfilled much later. Overall, choosing 8 threads to explore energy-efficient mappings for a given accelerator is a reasonable trade-off to achieve low simulation runtime.

The runtime of the measurement-based node evaluation clearly depends on the performance of the hardware components deployed and the number of concurrent threads for benchmarking. Hence, Table 3 shows the median runtime to obtain latency of CNN inference for all partitioning points, where PyTorch uses 8 threads on the Intel Xeon W-2145 and 32 threads on the AMD EPYC 7702P. The benchmarking is performed 500 times to ensure a certain reliability of the results. Most importantly, compared to our previous work where we measured a median runtime of 3320 seconds for the FCN ResNet-50 on a Jetson TX-2 for 1000 runs [6], we can achieve a significant improvement in the overall simulation time with the reduced number of partitioning points.

Table 3: Median time and standard deviation to measure latency of all potential partitions for different CNN architectures and two CPUs out of 10 runs. The measurement-based node evaluation includes 500 runs for each element in the set of partitioned CNN models. The Intel Xeon W-2145 uses 8 threads for benchmarking whereas the AMD EPYC 7702P runs 32 concurrent threads.

Architecture	**Partitioning points**	**Sensor Node**		**Edge Node**	
		Xeon W-2145	EPYC 7702P	Xeon W-2145	EPYC 7702P
FCN ResNet-50	10	342.42 s ($\sigma = 4.00\ s$)	294.97 s ($\sigma = 7.08\ s$)	367.95 s ($\sigma = 2.25\ s$)	322.85 s ($\sigma = 6.72\ s$)
GoogLeNet	13	127.88 s ($\sigma = 0.70\ s$)	114.06 s ($\sigma = 0.48\ s$)	84.09 s ($\sigma = 0.77\ s$)	66.67 s ($\sigma = 0.75\ s$)
SqueezeNet V1.1	8	25.46 s ($\sigma = 1.09\ s$)	24.88 s ($\sigma = 0.79\ s$)	15.51 s ($\sigma = 1.16\ s$)	16.49 s ($\sigma = 0.39\ s$)

4.3. Experimental Results

Besides analyzing a given CNN architecture and identifying a set of potential partitioning points to evaluate, CNNParted also outputs various metrics to support the design process. In the following, we therefore provide insights into determining beneficial partitioning points and system configurations based on the evaluation results of CNNParted.

4.3.1. Ethernet Link

The link is a critical part of the entire system, as it not only has to transfer the data of a single application from the sensor to the edge node, but also has to serve multiple processes at the same time. Hence, reducing traffic is an important optimization criterion to allow multiple applications to run simultaneously. In order to achieve lower link utilization, CNNParted provides insights into the size of the output feature map of each possible partitioning point. We will use SqueezeNet V1.1 topology and Gigabit EEE with a configuration of $\sigma_{off} = 0.1$ in the following to evaluate the impact of the link to the entire system.

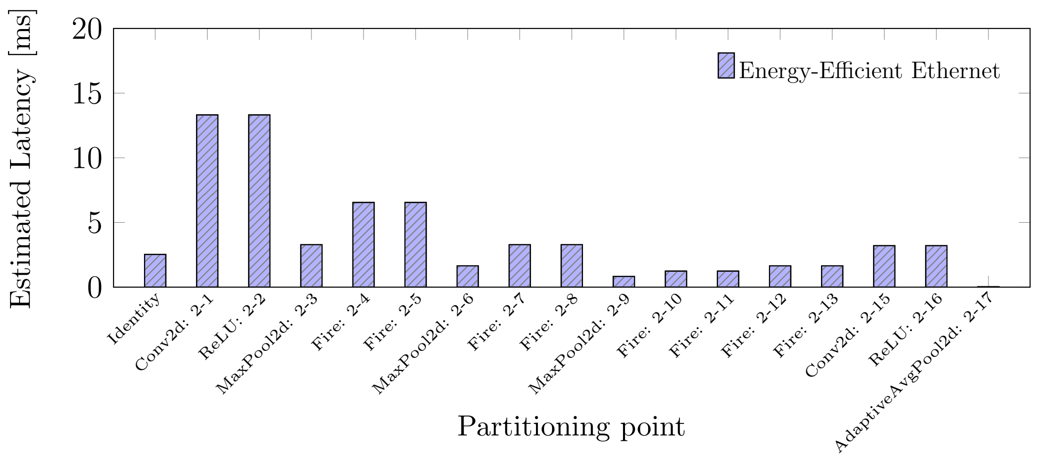

Figure 4: SqueezeNet V1.1 evaluation results of each potential partitioning point for unconstrained output feature map size regarding the link latency. The partitioning points *Conv2d: 2-1* and *Fire: 2-4* require large bandwidth and are therefore disadvantageous. In contrast, *MaxPool2d: 2-9* is favorable in terms of latency if the entire CNN is not to be processed in the sensor node alone.

As shown in Figure 4, when searching for a partitioning point that is also beneficial in terms of the required link bandwidth, several split points can already be sorted out from the very beginning. Especially the second (*Conv2D: 2-1*) and third point(*ReLu: 2-2*) as well as the partitioning points *Fire: 2-5* and *Fire: 2-6* require long time for transmitting the data. Therefore, these should be avoided when optimizing towards reduced traffic on the link.

Apart from bandwidth considerations, our experimental results for transmitting 25 frames per second show significantly lower energy consumption for EEE compared to conventional Ethernet, as expected, while the latency overhead of EEE is only about 0.01 ms. Exemplary, energy consumed for sending intermediate results after processing *Fire: 2-11* is reduced from 0.74 mJ to 0.098 mJ.

4.3.2. Sensor Node Architecture

CNNParted not only aims to find suitable partitioning points in a static system, but can also be used to explore different hardware architectures based on a given workload. Thus, using the model-based approach for evaluation also allows the framework to be used during the design phase of a System-on-Chip (SoC). In the following, we therefore investigate the performance of each hardware accelerator for the GoogLeNet topology.

Figure 5 shows the estimated latency for each potential partitioning point using one of the two hardware accelerators. Since the Simba-like architecture is optimized towards performance and runs at 500 MHz, while the Eyeriss-like accelerator is clocked at 200 MHz and optimized for low energy consumption, the results are as expected. However, if power consumption is also taken into account, the choice is no longer trivial.

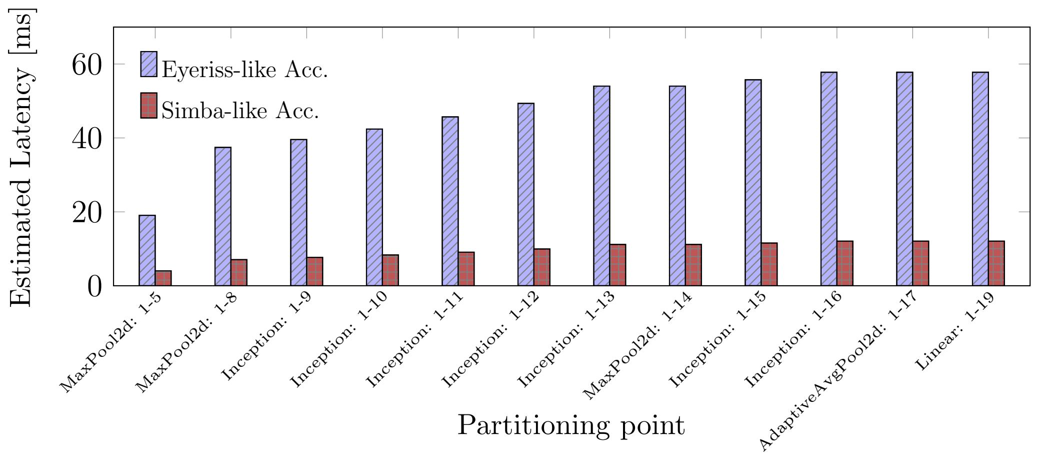

Figure 5: GoogLeNet evaluation results of each potential partitioning point using either Eyeriss-like (clocked at 200 MHz) or Simba-like architecture (clocked at 500 MHz). Obviously, since the clock frequencies are not equal, Simba-like accelerator can run inference faster. Besides, it can be observed that the first layers up to the partitioning point *Inception: 1-12* have the largest contribution to the latency.

Based on the estimated power that can be obtained from the simulation results of CNNParted, the power consumption of these accelerators is about 200 mW for the Eyeriss-like architecture and about 650 mW for the Simba-like accelerator, respectively. Thus, if the system allows the latency of the Eyeriss-like architecture, but higher data throughput is required, a second accelerator could be implemented to increase throughput without consuming more power than a single Simba-like architecture. Moreover, from the simulation results, we can deduce that especially the rear CONV layers in GoogLeNet do not contribute much to the latency. Therefore, combining both accelerators within the sensor node may be a feasible solution if the SoC provides enough area. Consequently, it would be beneficial to process the first CONV layers in the Simba-like accelerator while the remaining layers until the partitioning point can be handled by the eyeriss-like accelerator. In this case, an advantageous data flow would be to infer the first CONV layers in the Simba-like accelerator, while the remaining layers up to the partitioning point are processed by the Eyeriss-like accelerator. This approach could thus enable high throughput and reduce overall power consumption compared to running all layers on the Simba-like accelerator.

4.3.3. System Evaluation

As an extension of our previous work [6] evaluating CNNs for a system consisting of a dedicated hardware accelerator integrated into the sensor and an NVIDIA Jetson TX-2 GPU as the edge node, CNNParted allows model-based evaluation in both parts of the system. CNNParted thus enables a holistic, model-based evaluation of the hardware accelerators deployed in the system as part of the hardware/software co-design. Below, we evaluate the energy consumption and latency for GoogLeNet inference partitioning for a system consisting of an Eyeriss-like SoC close to the sensor and a Simba-like hardware accelerator in the central computing node. The results are shown in Figure 6.

Looking at the overall energy consumption of the system, shown in Figure 6a, it can be seen that either partitioning right at the beginning or at the end of the network is best. These points also offer beneficial results in terms of latency and required link bandwidth, respectively. In between, there is a point of interest with the partitioning point *MaxPool2d: 1-14* due to its low bandwidth requirements. Compared to the transmission of pure raw data, this point offers a bandwidth reduction of almost 73 %. However, it suffers from a relatively high latency, as depicted in Figure 6b, therefore *MaxPool2d: 1-14* should only be selected if this is not an important optimization criterion. *MaxPool2d: 1-5* offers a good workload distribution, since the difference in latency between the two nodes is the lowest of all the partitioning points, except for the discarded ones. In particular, *MaxPool2d: 1-2* and *BasicConv2d: 1-3* offer low overall latency but exceed the link constraint of 200,000 elements in the feature map. However, both layers output a feature map with only 704 elements above the user-defined limit as stated before, clearly demonstrating the need for careful constraint definition. Nevertheless, the comparatively quite high load on the link when partitioning at these points (+33 % compared to transmitting raw data) or at *MaxPool2d: 1-5* can be problematic in some use cases. Finally, a good trade-off between required

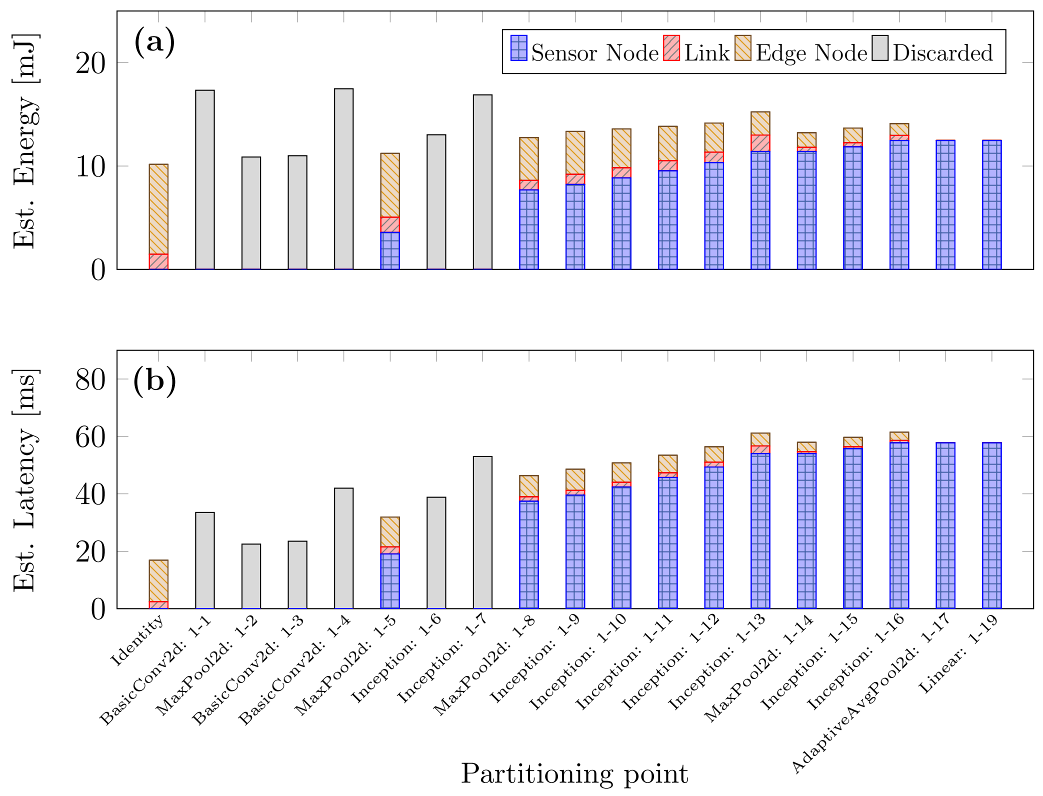

Figure 6: GoogLeNet evaluation results of each possible partitioning point using an Eyeriss-like architecture clocked at 200 MHz in the sensor and a Simba-like architecture clocked at 500 MHz in the edge node. If minimal latency is required, the network has to be executed entirely on the edge node. Partitioning point *Inception: 1-9* provides a good trade-off in terms of power consumption, latency, required bandwidth and workload distribution.

bandwidth, latency and energy consumption is provided by the partitioning points *MaxPool2d: 1-8* and *Inception: 1-9*.

In general, it can be observed that the partitioning of CNNs after a pooling layer is beneficial because these can reduce the overall size of the feature maps and thus the amount of data transferred. Moreover, since such layers are not compute-intensive, sensor latency does not increase drastically, unlike connection latency and energy consumption which may benefit significantly.

4.3.4. Baseline Comparison

As mentioned in the state of the art analysis, there are tools such as DeepCOD proposed by Yao et al. [17] that enable offloading DNN inference from a central compute node to near-sensor devices to reduce network latency. For comparison with their presented results, we evaluate the FCN-ResNet-50 in terms of latency and energy consumption for each potential partitioning point. n the following, we focus on the first layers up to the *Bottleneck: 3-7*, since the subsequent suitable partitioning points are not feasible in the context of resource-constrained sensor nodes due to the large number of layer parameters. The results for using a Simba-like hardware accelerator architecture in the sensor node and an AMD EPYC 7702P core in the edge node are shown in Figure 7.

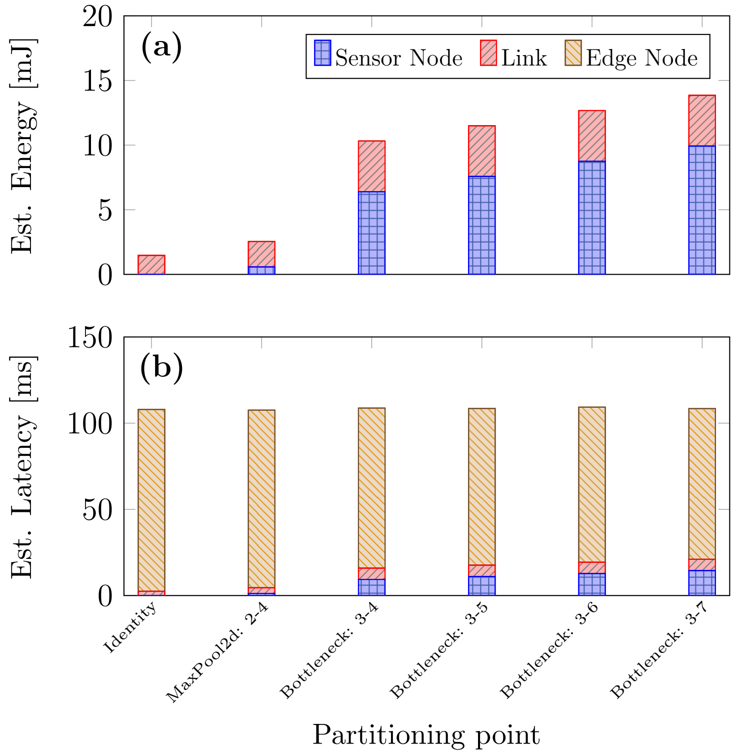

Figure 7: FCN-ResNet50 evaluation results of each possible partitioning point using an Simba-like architecture clocked at 500 MHz in the sensor and an AMD EPYC 7702P in the edge node. If low energy consumption and network latency is targeted, partitioning point *MaxPool2d: 2-4* provides a good trade-off.

For finding a beneficial partitioning point DeepCOD takes inference latency and accuracy into account. The latter is necessary, since the framework adds an encoder-decoder structure to the DNN to further reduce the required bandwidth for transmitting intermediate feature maps. However, near-sensor platforms are often limited in their energy consumption. As can be seen in Figure 7a, neglecting energy consumption when determining the best partitioning point is therefore not reasonable. Especially in this case, where latency is not impacted by choosing different partitioning points depending on the system requirements of offloading DNN inference as shown in Figure 7b, energy consumption becomes a major metric for determining the best partitioning point. Apart from determining a beneficial partitioning point, CNNParted, unlike DeepCOD and similar tools, does not insert additional encoder-decoder structures into the DNN architecture and therefore does not require training to achieve high network accuracy.

5. Conclusion

Efficient CNN inference partitioning in embedded systems for applications such as autonomous driving or assistive robotics requires a comprehensive hardware/software co-design. Therefore in this paper, we presented CNNParted, an open source framework for hardware-aware design space exploration of CNN partitioning. Based on simulation and measurements, it estimates latency and energy consumption for each potentially beneficial partitioning point supporting the system designer determining an optimal workload distribution between near sensor node and central compute platform. Thereby, not only the hardware architecture deployed in the nodes is considered but also the impact of the link on the system. Hence, CNNParted evaluates CNN inference partitioning considering the whole system architecture with respect to the available link bandwidth.

The evaluation results presented proved the effectiveness of CNNParted to find multiple points of interest for three, commonly used CNNs. Beyond identifying beneficial partitioning points for FCN ResNet-50, GoogLeNet, and SqueezeNet V1.1, we were also able to derive important metrics such as latency and energy consumption for a holistic hardware/software co-design.

CNNParted has been designed to explore inference partitioning for a given CNN considering the system hardware configuration and to support the system design process. In the future, we plan to integrate it into Neural Architecture Search (NAS) to account for the system hardware setup in an earlier stage of application development. In addition, we further plan to evaluate CNN partitioning using wireless links such as Bluetooth and FPGA-accelerated inference in heterogeneous sensor systems.

Acknowledgment

SPONSORED BY THE

Federal Ministry of Education and Research

This work was funded by the German Federal Ministry of Education and Research (BMBF) under grant number 16ME0454 (EMDRIVE). The responsibility for the content of this publication lies with the authors.

References

[1] N. Fasfous, M.-R. Vemparala, A. Frickenstein, M. Badawy, F. Hundhausen, J. Höfer, N.-S. Nagaraja, C. Unger, H.-J. Vögel, J. Becker, T. Asfour, W. Stechele, Binary-lorax: Low-latency runtime adaptable xnor classifier for semi-autonomous grasping with prosthetic hands, in: 2021 IEEE International Conference on Robotics and Automation (ICRA), 2021, pp. 13430–13437. doi:10.1109/ICRA48506.2021.9561045.

[2] S. Bianco, R. Cadene, L. Celona, P. Napoletano, Benchmark analysis of representative deep neural network architectures, IEEE Access 6 (2018) 64270–64277. doi:10.1109/ACCESS.2018.2877890.

[3] S. Han, et al., EIE: efficient inference engine on compressed deep neural network, CoRR abs/1602.01528 (2016). arXiv:1602.01528.

[4] I. Walter, J. Ney, T. Hotfilter, V. Rybalkin, J. Hoefer, N. Wehn, J. Becker, Embedded face recognition for personalized services in the assistive robotics, in: Machine Learning and Principles and Practice of Knowledge Discovery in Databases, Springer International Publishing, Cham, 2021, pp. 339–350.

[5] T. Hotfilter, F. Kempf, J. Becker, D. Reinhardt, I. Baili, Embedded image processing the european way: A new platform for the future automotive market, in: 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), 2020, pp. 1–6. doi:10.1109/WF-IoT48130.2020.9221396.

[6] F. Kreß, J. Hoefer, T. Hotfilter, I. Walter, V. Sidorenko, T. Harbaum, J. Becker, Hardware-aware partitioning of convolutional neural network inference for embedded ai applications, in: 2022 18th International Conference on Distributed Computing in Sensor Systems (DCOSS), 2022, pp. 133–140. doi:10.1109/DCOSS54816.2022.00034.

[7] Y. Chen, Y. Xie, L. Song, F. Chen, T. Tang, A survey of accelerator architectures for deep neural networks, Engineering 6 (3) (2020) 264–274. doi:https://doi.org/10.1016/j.eng.2020.01.007.

[8] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, L. Zhong, Redeye: Analog convnet image sensor architecture for continuous mobile vision, SIGARCH Comput. Archit. News 44 (3) (2016) 255–266. doi:10.1145/3007787.3001164.

[9] C. Hu, W. Bao, D. Wang, F. Liu, Dynamic adaptive dnn surgery for inference acceleration on the edge, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 1423–1431. doi:10.1109/INFOCOM.2019.8737614.

[10] W. Zhang, J. Zhang, M. Shen, G. Luo, N. Xiao, An efficient mapping approach to large-scale dnns on multi-fpga architectures, in: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2019, pp. 1241–1244. doi:10.23919/DATE.2019.8715174.

[11] D. Kwon, S. Hur, H. Jang, E. Nurvitadhi, J. Kim, Scalable multi-fpga acceleration for large rnns with full parallelism levels, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6. doi:10.1109/DAC18072.2020.9218528.

[12] T. Mohammed, C. Joe-Wong, R. Babbar, M. D. Francesco, Distributed inference acceleration with adaptive dnn partitioning and offloading, in: IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, 2020, pp. 854–863. doi:10.1109/INFOCOM41043.2020.9155237.

[13] T. Alonso, et al., Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning, ACM Trans. Reconfigurable Technol. Syst. 15 (2) (dec 2021). doi:10.1145/3470567.

[14] S. Teerapittayanon, B. McDanel, H. Kung, Distributed deep neural networks over the cloud, the edge and end devices, in: 2017 IEEE 37th

International Conference on Distributed Computing Systems (ICDCS), 2017, pp. 328–339. doi:10.1109/ICDCS.2017.226.

[15] M. Ghasemi, S. Heidari, Y. G. Kim, A. Lamb, C.-J. Wu, S. Vrudhula, Energy-efficient mapping for a network of dnn models at the edge, in: 2021 IEEE International Conference on Smart Computing (SMARTCOMP), 2021, pp. 25–30. doi:10.1109/SMARTCOMP52413.2021.00024.

[16] J. H. Ko, T. Na, M. F. Amir, S. Mukhopadhyay, Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained internet-of-things platforms, in: 2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), 2018, pp. 1–6. doi:10.1109/AVSS.2018.8639121.

[17] S. Yao, J. Li, D. Liu, T. Wang, S. Liu, H. Shao, T. Abdelzaher, Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency, in: Proceedings of the 18th Conference on Embedded Networked Sensor Systems, SenSys '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 476–488. doi:10.1145/3384419.3430898.

[18] D. Hu, B. Krishnamachari, Fast and accurate streaming cnn inference via communication compression on the edge, in: 2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI), 2020, pp. 157–163. doi:10.1109/IoTDI49375.2020.00023.

[19] Z. Zhao, K. M. Barijough, A. Gerstlauer, Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (11) (2018) 2348–2359. doi:10.1109/TCAD.2018.2858384.

[20] L. Yang, W. Jiang, W. Liu, E. H. M. Sha, Y. Shi, J. Hu, Co-exploring neural architecture and network-on-chip design for real-time artificial intelligence, in: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), 2020, pp. 85–90. doi:10.1109/ASP-DAC47756.2020.9045595.

[21] S. Disabato, M. Roveri, C. Alippi, Distributed deep convolutional neural networks for the internet-of-things, IEEE Transactions on Computers 70 (8) (2021) 1239–1252. doi:10.1109/TC.2021.3062227.

[22] E. Shelhamer, J. Long, T. Darrell, Fully convolutional networks for semantic segmentation, arXiv:1605.06211 [cs]ArXiv: 1605.06211 (May 2016).

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems 32, Curran Associates, Inc., 2019, pp. 8024–8035.

[24] K. Guo, S. Zeng, J. Yu, Y. Wang, H. Yang, A survey of fpga-based neural network accelerator (2018). arXiv:1712.08934.

[25] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, H. Yang, Going deeper with embedded fpga platform for convolutional neural network, in: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 26–35. doi:10.1145/2847263.2847265.

[26] B. Khabbazan, S. Mirzakuchaki, Design and implementation of a low-power, embedded cnn accelerator on a low-end fpga, in: 2019 22nd Euromicro Conference on Digital System Design (DSD), 2019, pp. 647–650. doi:10.1109/DSD.2019.00102.

[27] A. Jahanshahi, Tinycnn: A tiny modular cnn accelerator for embedded fpga (2019). doi:10.48550/ARXIV.1911.06777.

[28] J. Kim, J.-K. Kang, Y. Kim, A resource efficient integer-arithmetic-only fpga-based cnn accelerator for real-time facial emotion recognition, IEEE Access 9 (2021) 104367–104381. doi:10.1109/ACCESS.2021.3099075.

[29] K. Zeng, Q. Ma, J. W. Wu, Z. Chen, T. Shen, C. Yan, Fpga-based accelerator for object detection: A comprehensive survey, The Journal of Supercomputing (2022) 1–41.

[30] NVIDIA Corporation, The nvidia deep learning accelerator (Nov. 2018). URL `http://nvdla.org/`

[31] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, et al., Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures, arXiv preprint arXiv:1911.09925 3 (2019).

[32] Y.-C. Lee, T.-S. Hsu, C.-T. Chen, J.-J. Liou, J.-M. Lu, Nnsim: A fast and accurate systemc/tlm simulator for deep convolutional neural network accelerators, in: 2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 2019, pp. 1–4. `doi:10.1109/VLSI-DAT.2019.8741950`.

[33] S. Kim, J. Wang, Y. Seo, S. Lee, Y. Park, S. Park, C. S. Park, Transaction-level model simulator for communication-limited accelerators (2020). `doi:10.48550/ARXIV.2007.14897`.

[34] T. Hotfilter, J. Hoefer, F. Kreß, F. Kempf, J. Becker, Flecsim-soc: A flexible end-to-end co-design simulation framework for system on chips, in: 2021 IEEE 34th International System-on-Chip Conference (SOCC), 2021, pp. 83–88. `doi:10.1109/SOCC52499.2021.9739212`.

[35] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, N. D. Lane, Best of both worlds: Automl codesign of a cnn and its hardware accelerator, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6. `doi:10.1109/DAC18072.2020.9218596`.

[36] F. Muñoz-Martínez, J. L. Abellán, M. Acacio, T. Krishna, Stonne: A detailed architectural simulator for flexible neural network accelerators, ArXiv abs/2006.07137 (2020).

[37] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, T. Krishna, Scale-sim: Systolic cnn accelerator simulator, arXiv preprint arXiv:1811.02883 (2018).

[38] S. L. Xi, Y. Yao, K. Bhardwaj, P. Whatmough, G.-Y. Wei, D. Brooks, Smaug: End-to-end full-stack simulation infrastructure for deep learning workloads, ACM Trans. Archit. Code Optim. 17 (4) (Nov. 2020). doi:10.1145/3424669.

[39] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, M. Horowitz, Interstellar: Using halide's scheduling language to analyze dnn accelerators, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 369–383. doi:10.1145/3373376.3378514.

[40] L. Mei, P. Houshmand, V. Jain, S. Giraldo, M. Verhelst, Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators, IEEE Transactions on Computers 70 (8) (2021) 1160–1174. doi:10.1109/TC.2021.3059962.

[41] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, J. Emer, Timeloop: A systematic approach to dnn accelerator evaluation, in: 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2019, pp. 304–315. doi:10.1109/ISPASS.2019.00042.

[42] Y. N. Wu, J. S. Emer, V. Sze, Accelergy: An architecture-level energy estimation methodology for accelerator designs, in: 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019, pp. 1–8. doi:10.1109/ICCAD45719.2019.8942149.

[43] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, N. P. Jouppi, Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques, in: 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2011, pp. 694–701. doi:10.1109/ICCAD.2011.6105405.

[44] Y. S. Shao, B. Reagen, G. Wei, D. Brooks, Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures, in: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 2014, pp. 97–108. doi:10.1109/ISCA.2014.6853196.

[45] S. Herreria-Alonso, M. Rodriguez-Perez, M. Fernandez-Veiga, C. Lopez-Garcia, A gi/g/1 model for 10 gb/s energy efficient ethernet links, IEEE Transactions on Communications 60 (11) (2012) 3386–3395. doi:10.1109/TCOMM.2012.081512.120089.

[46] P. Fondo-Ferreiro, M. Rodríguez-Pérez, M. Fernández-Veiga, Implementing energy saving algorithms for ethernet link aggregates with onos, in: 2018 Fifth International Conference on Software Defined Systems (SDS), 2018, pp. 118–125. doi:10.1109/SDS.2018.8370432.

[47] IEEE, Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 3: Csma/cd access method and physical layer specifications amendment 5: Media access control parameters, physical layers, and management parameters for energy-efficient ethernet, IEEE Std 802.3az-2010 (Amendment to IEEE Std 802.3-2008) (2010) 1–302doi:10.1109/IEEESTD.2010.5621025.

[48] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, S. W. Keckler, Simba: Scaling deep-learning inference with multi-chip-module-based architecture, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, p. 14–27. doi:10.1145/3352460.3358302.

[49] Y.-H. Chen, T.-J. Yang, J. Emer, V. Sze, Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices, IEEE Journal on Emerging and Selected Topics in Circuits and Systems 9 (2) (2019) 292–308. doi:10.1109/JETCAS.2019.2910232.

[50] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, arXiv:1409.4842 [cs]ArXiv: 1409.4842 (Sep 2014).

[51] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer, Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, arXiv:1602.07360 [cs]ArXiv: 1602.07360 (Nov 2016).