Noname manuscript No.
(will be inserted by the editor)

AMAIX in-depth: A Generic Analytical Model for Deep Learning Accelerators

Niko Zurstraßen · Lukas Jünger · Tim Kogel · Holger Keding · Rainer Leupers

Received: date / Accepted: date

Abstract In recent years the growing popularity of Convolutional Neural Networks (CNNs) has driven the development of specialized hardware, so called Deep Learning Accelerators (DLAs). The large market for DLAs and the huge amount of papers published on DLA design show that there is currently no one-size-fits-all solution. Depending on the given optimization goals such as power consumption or performance, there may be several optimal solutions for each scenario. A commonly used method for finding these solutions as early as possible in the design cycle, is the employment of analytical models which try to describe a design by simple yet insightful and sufficiently accurate formulas. The main contribution of this work is the generic Analytical Model for AI accelerators (AMAIX) for the estimation of CNN execution time on DLAs. It is based on the popular Roofline model. To show the validity of our approach, AMAIX was applied to the Nvidia Deep Learning Accelerator (NVDLA) as a case study using the AlexNet and LeNet CNNs as workloads. The resulting performance predictions were verified against an RTL emulation of the NVDLA using a Synopsys ZeBu Server-based hybrid prototype. By refining the model following a divide-and-conquer paradigm, AMAIX predicted the inference time of AlexNet and LeNet on the NVDLA with an accuracy 98%. Furthermore, this work shows how to use the obtained results for root-cause analysis and as a starting point for design space exploration.

Keywords Deep Learning Accelerators · Analytical Models · Design Space Exploration · Roofline Model

Niko Zurstraßen
ICE RWTH Aachen University
E-mail: zurstrassen@ice.rwth-aachen.de

Lukas Jünger
ICE RWTH Aachen University
E-mail: juenger@ice.rwth-aachen.de

1 Introduction

CNNs have become part of our everyday life in the last years. They can be found in many different applications such as smartphones, data centers, and autonomous driving systems [4,13,15]. These different applications have varying requirements regarding their Key Performance Indicators (KPIs) such as performance, area, power consumption, or accuracy. Executing CNNs on general purpose CPUs does often not fulfill these requirements. Therefore, specialized integrated circuits, called Deep Learning Accelerators (DLAs), are developed to mitigate this issue. Designing a DLA is not a trivial task, since a DLA usually consists of different acceleration units for different CNN layers. Each of these units has configuration parameters, leaving the designer with a large design space to explore. For example, the number of Multiply-Accumulate (MAC) units in an accelerator directly influences area, cost, power consumption, and performance. As this work shows, the optimal configuration of a DLA regarding performance strongly depends on the prospective workload, which should therefore also be considered from the very beginning. All these considerations should be included as early as possible in the development process. Although parameters can be altered in later stages as well, small modifications usually entail a series of further changes, especially if the design is already in a more advanced development stage. The more advanced the design is, the higher the costs for the alterations will be.

Common methods to perform early estimates are high-level architecture simulations, also called pre-RTL simulations, and analytical models. Both methods have been applied in DLA design [3,5,6,14,12]. However, most approaches focus strongly on the hardware structure to be developed and merely regard the model as a byproduct. These models are usually very specific and therefore cannot be generalized for designing new DLAs. A more generic approach is the *Eyexam* framework proposed in [5]. It shows how a performance evaluation for an arbitrary DLA can be created within seven refinement steps. However, the authors only provide a general overview of how these steps have to be applied and do not mention any formulas or further instructions.

In contrast, this paper presents a novel generic analytical model to estimate the inference performance of an arbitrary DLA. It is based on the popular Roofline model since the assumptions of data/processing parallelism and a small control flow overhead hold valid for most DLA designs [16]. The model still requires characterization regarding the DLA's hardware architecture, but provides a structured and systematic approach to attain it. Other KPIs such as power or area are left for future work. As a case study the model is applied to the Nvidia Deep Learning Accelerator (NVDLA), which was chosen because its open-source RTL implementation and compiler permit to verify the model in great detail. Hence, the estimated inference performance is compared to the results obtained by executing the unmodified NVDLA RTL code in a hybrid prototype using Synopsys ZeBu Server. In addition the estimates are compared to the official Nvidia NVDLA performance sheet [2]. Note, this work is an extended version of the original paper [8] published at "SAMOS In-

ternational Conference on Embedded Computer Systems: Architectures". The major contributions of the original work are as follows:

- The broadly-applicable AMAIX model for inference performance estimation of DLAs
- Detailed case study on how to apply AMAIX using the NVDLA
- Evaluation of AMAIX's accuracy using hybrid emulation
- Assessment of AMAIX for NVDLA design space exploration

We extend the original work by discussing the analytical model in greater detail and by proposing a divide-and-conquer paradigm to increase the model's accuracy.

2 The AMAIX Approach

This section deals with the main contribution of this work: AMAIX, a generic analytical model for predicting DLA inference time.
The fundament of AMAIX is the popular Roofline model [16] by Williams et al. Originally designed for the performance evaluation of multicore architectures, we extend the model to DLAs and show its validity. The key idea behind the Roofline model is that the achievable performance of a workload on a compute system is either limited by the available memory bandwidth or by the theoretical maximum compute power. To determine the limiting resource, one has to calculate the so-called operational intensity for a given task. The operational intensity is the ratio of number of operations divided by the number of bytes exchanged with the main memory for a given workload. By inserting the operational intensity into a roofline graph, as depicted in Fig. 1, the achievable performance can quickly be obtained by only visual means.
An assumption of the Roofline model is that the operational intensity is constant during the execution of a workload, and that the memory and processing resources used do not change. This is depicted in Fig. 1a), where the memory-bound `cnn0` task is modeled. If one of these conditions is not fulfilled, a task can be divided into further subtasks, which are mapped to different resources and can have different operational intensities. This is depicted in Fig. 1b). Here the main workload `cnn0` was split into the different tasks `layer0`, `layer1`, and `layer2`. `layer0` is memory bound by the peak bandwidth ceiling `memory1`, `layer1` is compute bound by peak performance ceiling `proc1`, and `layer2` is bound by `proc0`.

Representing an entire CNN as a single task, as for example in [13], was shown to be too simplistic and imprecise [7]. Usually the first convolutional layers of a CNN require only a few weights but many MAC operations, thus yielding a very high operational intensity. For the last, usually fully-connected layers, however, it is vice versa. These layers require many weights for only a few MAC operations. This was also confirmed by the experiments conducted in our case study. For example, the first convolutional layers of LeNet running on the NVDLA exhibit an operational intensity of 100 and more, while the

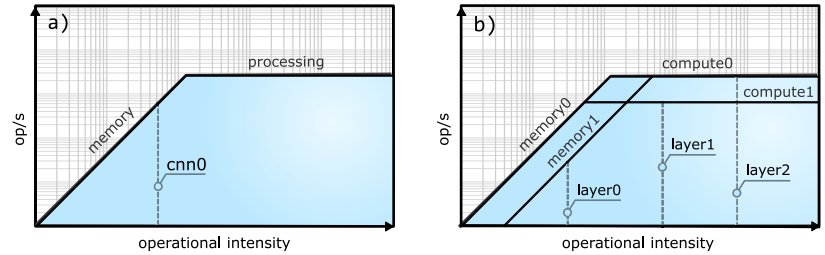

Fig. 1 a) The whole CNN is represented by one task which is mapped to one memory and one processing unit. b) The CNN is described on a layer level. Different layers expose different operational intensities and can be mapped to different memories and processing units.

fully-connected layers are in the range of 1 to 2. Therefore, the individual layers that form a CNN must be modeled as individual tasks. In addition, these tasks are usually mapped to different processing units on the DLA. For example on the NVDLA, convolutional layers are executed on the `CONV_CORE` processing unit, while pooling layers are executed on the `PDP` processing unit.

In AMAIX we propose that the amount of memory transfers, the number of arithmetic operations and the hardware resources used must be determined per CNN layer. For this, a mathematical description of CNN layer is introduced as follows:

$$
\begin{aligned}
l = (i,\, k,\, o,\, map,\, scale_{ifmap},\, scale_{weight},\, scale_{ofmap},\, scale_{ops}) \\
i = (i_w, i_h, i_c),\; k = (k_w, k_h, k_c, k_n),\; o = (o_w, o_h, o_c) \\
map \in \{(proc0, mem0),\, (proc1, mem1),\, (proc2, mem1),\, ...\} \\
scale_{ifmap} \;:\; map \times i \times k \times o \to \mathbb{R} \\
scale_{ofmap} \;:\; map \times i \times k \times o \to \mathbb{R} \\
scale_{weight} \;:\; map \times i \times k \times o \to \mathbb{R} \\
scale_{ops} \;:\; map \times i \times k \times o \to \mathbb{R}
\end{aligned}
$$

Here, i represents the dimensions (width, height, channels) of the input feature map (ifmap), o the dimensions of the output feature map (ofmap) and k the dimensions and number of kernels which are required for a layer's execution. Applying this formalization to LeNet's first layer would result in the following sets:

$$i = (28, 28, 1),\; k = (5, 5, 1, 20),\; o = (24, 24, 20)$$

Fig. 2 provides an illustration of these parameters. The *map* parameter specifies on which hardware resources a layer is executed. The scaling factors are functions which map a layer's parameters to a real number to incorporate the microarchitectural design of the DLA. They indicate how much the examined data transfers or arithmetic operations deviate from a general model. Since determining the scaling factors correctly is paramount for achieving high

modeling accuracy, a more detailed explanation is given in the following subsections.

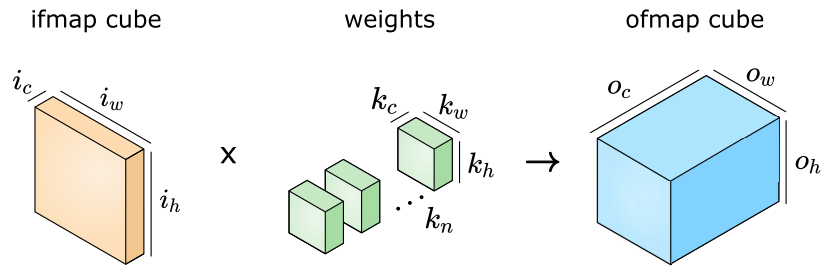

Fig. 2 Visual representation of ifmap, kernel and ofmap parameters.

2.1 Determining Data Transfers

The amount of all data transfers (d_{total}) for a CNN's layer is the sum of ifmap data (d_{ifmap}), weight data (d_{weight}), and ofmap data (d_{ofmap}):

$$
\begin{aligned}
d_{total} &= d_{ifmap} + d_{weight} + d_{ofmap} \\
d_{ifmap} &= scale_{ifmap} \cdot i_w \cdot i_h \cdot i_c \\
d_{weight} &= scale_{weight} \cdot k_w \cdot k_h \cdot k_c \cdot k_n \\
d_{ofmap} &= scale_{ofmap} \cdot o_w \cdot o_h \cdot o_c
\end{aligned}
$$

If $scale_{ifmap} = scale_{weight} = scale_{ofmap} = 1$ is used, the general model is assumed. For example, according to this model the ifmap data is just the number of ifmap elements at one byte per element. This is the volume of the ifmap cuboid shown in Fig. 2. The general model is a good starting point for initial estimates and can be used when there is little information available about the actual hardware.

In practice, there are a number of effects depending on the DLA microarchitecture and executed algorithms causing a scaling smaller or larger than 1. The following list gives an overview of influences on the data scaling factors:

- **Data reload:** On many systems, the size of the on-chip memory is not sufficient to buffer the entire ifmap, kernel and ofmap. This means that the same data has to be fetched/written multiple times from/to the main memory causing an increased scaling factor.
- **Data type:** Frequently used data types are, for example, `int8` (1 B) or `fp16` (2 B). This must be considered accordingly.
- **Dark bandwidth:** When transferring data via a bus system, the size of the data must be a multiple of the bus width. If this is not the case, dark bandwidth occurs, which results in a larger scaling factor.

- **Zero-padding:** The internal word width of a DLA can cause the data to be padded with zeros increasing the scaling factor.
- **Transformation:** This applies in particular to convolution operations, which can not only be implemented by the standard algorithm. Fourier transform, Winograd convolution, or Toeplitz matrices, can influence the scaling factors.
- **Layer fusion:** Since the output of one layer is usually the input of another, data can be kept locally, which allows a data scaling factor of 0.
- **Data compression**: Data can be compressed resulting in a smaller scaling factor.

2.2 Determining the Number of Operations

Similar to determining data transfers, a formula for the number of arithmetic operations for a CNN's layer is derived:

$$n_{ops} = scale_{ops} \cdot o_w \cdot o_h \cdot o_c \cdot k_w \cdot k_h \cdot k_c$$

For $scale_{ops} = 1$ this formula refers to the number of MAC operations needed for a standard convolution and is also a good first order estimate if no knowledge about the hardware is available. Implementation details of hardware and algorithms can increase or decrease the number of operations scaling factor $scale_{ops}$. Two effects play a particularly important role:

- **Transformation:** Alternative convolution algorithm implementations like Fourier transform or Winograd convolution usually decrease the amount of needed operations.
- **Hardware utilization:** Many DLA designs have fixed processing engine sizes resulting only in a 100% utilization if the data's dimensions comply with these sizes. Chen et. al. distinguish between the two cases of *spatial mapping fragmentation* and *temporal mapping fragmentation* leading to underutilized hardware [5]. Since both play an important role in most DLAs, the NVDLA case study section provides an in-depth explanation on how to quantify this effect.

After determining all the scaling factors, a detailed Roofline model can be created. This is covered in the next subsection.

2.3 Applying the Roofline Model

In this subsection the previously presented assumptions and formulas are joined together. As a first step, the Roofline model must be reformulated for each layer l of the CNN L as follows:

$$performance(l) = \min(\, performance_{peak}(l),\, op_{intensity}(l) \cdot memory_{peak}(l) \,)$$

$$op_{intensity}(l) = \frac{n_{ops}(l)}{d_{total}(l)}$$

The inference time of a CNN is the sum over all layer time spans t_{layer}:

$$t_{layer}(l) = n_{ops}(l)/performance(l)$$
$$t_{total}(L) = \sum_{l \in L} t_{layer}(l)$$

Another aspect to be considered is the pipelining of layer operators. Many DLAs like the NVDLA are systolic architectures on layer-level. If one or more layers are pipelined, they must be considered as a whole. The following formulas then apply for a pipeline of layers $pipe = \{l_n, ..., l_{n+m}\}$:

$$t_{layer}(pipe) = \max\left(\frac{n_{ops}(l_n)}{performance(l_n)}, ..., \frac{n_{ops}(l_{n+m})}{performance(l_{n+m})}\right)$$
$$op_{intensity}(l_{dom}) = \frac{n_{ops}(l_{dom})}{\sum_{k \in pipe} d_{total}(k)}$$

Note, that this model assumes that the overhead for filling and draining a pipeline can be omitted. It can be observed that the slowest unit in a pipeline determines the overall execution time and therefore the performance. A layer l_{dom} which determines a pipeline's executions time is called *dominating*. With all the formulas and descriptions listed above the model is now ready to be applied to an example.

3 Case Study: Nvidia Deep Learning Accelerator

In this section AMAIX, as presented in the preceding section, is applied to the NVDLA. The key challenge here is to determine the different scaling factors. This is done for bias and convolutional layers as examples in the following. For other layers only the results are presented since a detailed description would go beyond the scope of this work. With these scaling factors the inference time of the NVDLA is estimated for the widely-used AlexNet and LeNet CNNs [10, 11]. These times are then compared with the results of an NVDLA Verilog emulation running in a hybrid prototype based on Synopsys ZeBu Server and Virtualizer. Finally, it is shown how AMAIX can be refined and used to explore the NVDLA's design space.

3.1 Nvidia Deep Learning Accelerator

The NVDLA is an open-source DLA specialised in CNN inference [2]. The project, which exists since 2017, features an open-source SystemC model, a Verilog implementation as well as a corresponding Kernel Mode Driver (KMD) and User Mode Driver (UMD). Executables for the NVDLA can be generated by using the NVDLA compiler. The NVDLA has over 30 configurable hardware parameters. One predefined configuration is the so called NVDLA full

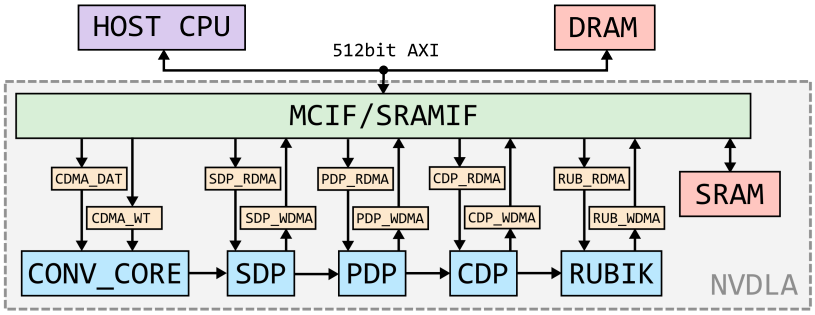

Fig. 3 Overview of the NVDLA full configuration.

configuration, which is used in this work since it contains all subprocessors and extensions. Fig. 3 shows an overview of the NVDLA full configuration.

It can be observed, that the NVDLA is composed of several specialized subprocessors for convolution (`CONV_CORE`), activation functions (`SDP`), pooling (`PDP`), normalization functions (`CDP`), and memory-to-memory transformations (`RUBIK`). Also it includes on-chip SRAM and a 512bit wide AXI bus interface. Data is fetched and written by dedicated DMAs for each subprocessor.

3.2 Hybrid Emulation Setup

To verify the results obtained from AMAIX, a hybrid prototype based on Synopsys ZeBu Server and Virtualizer was used for comparison. Here, the NVDLA RTL is synthesized for the ZeBu server and then emulated on it, meaning that precise behavioral analysis can be undertaken. In our hybrid emulation setup additional components such as an ARM Cortex A57 CPU cluster and DRAM are added to form an entire embedded system. Since these components only need to be modeled functionally they are part of a Virtualizer SystemC TLM2.0 Virtual Platform (VP) that is executed on a host computer. This is depicted in Fig. 4. VP and RTL emulation are connected via so-called transactors. Physically a PCIe bus is used for this purpose.

Inside the VP a Linux operating system with the NVDLA drivers is executed on the ARM cluster. To reduce the system's overhead, the simulated ARM cores were clocked at 4 GHz while the NVDLA was clocked at 1 GHz. The DRAM provided in the VP is purely functional and provides no timing annotation. Thus the NVDLA's bandwidth is limited only by its clock speed and bus width, which corresponds to 64 GB/s for the NVDLA full configuration at 1 GHz. This approximation was shown to be valid using a Synopsys Platform Architect Ultra pre-RTL simulation [9]. Using this simulation the DRAM access patterns of the NVDLA were analyzed. It was observed, that nearly 100% of the DRAM bandwidth can be utilized for weight fetching, which dominates the overall data traffic ($> 90\%$). This is due to the linear access pattern of the `CDMA_WT`, which is responsible for fetching the weights. The other DMAs

showed only partially linear patterns, which also reached over 95% depending on the DRAM and bus configuration in our simulations.

Using the mentioned hybrid emulation setup the execution time for most commonly-used networks like AlexNet or ResNet-18 on the emulated NVDLA is in the range of a few minutes. This allows us to analyze different scenarios quickly.

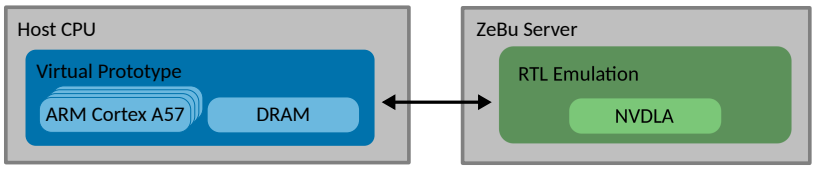

Fig. 4 Hybrid emulation setup.

3.3 Applying AMAIX

As a first example the scaling factors of a convolutional layer shall be derived. These layers are executed on the NVDLA's `CONV_CORE` which provides a maximum compute power of:

$$performance_{peak} = T_k \cdot T_c \cdot clock$$

With T_k being the width of the NVDLA's MAC unit (which is part of the `CONV_CORE`) and T_c being the depth of the MAC unit. The MAC unit implements a typical weight-stationary architecture which can also be found in other DLAs. For the NVDLA full configuration with a data type b of fp16, the parameters resolve to $T_k = 16$ and $T_c = 64$.

As a next step the operations scaling factor is derived as:

$$scale_{ops} = \left\lceil \frac{i_c}{T_c} \right\rceil \cdot \left\lceil \frac{k_n}{T_k} \right\rceil \cdot \frac{T_k \cdot T_c}{i_c \cdot k_n}$$

The formula incorporates the previously mentioned cases of *spatial mapping fragmentation* and *temporal mapping fragmentation*. A spatial mapping fragmentation occurs in case of the NVDLA if $i_c < T_c$ and $k_n < T_k$ apply. Temporal mapping fragmentation is similar, but refers to i_c and k_n not being multiples of T_c and T_k. This means that spatial mapping fragmentation never achieves a 100% hardware utilization while temporal mapping fragmentation achieves a 100% hardware utilization only in some cycles of the execution (see Fig. 5).

To model a lower hardware utilization one can either adjust the computational roof for a given layer or add dark operations. These are operations that are executed but do not contribute to the actual result. In this work the latter

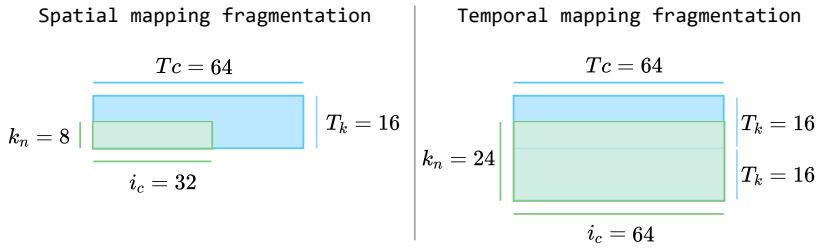

Fig. 5 Depicting temporal and spatial mapping fragmentation. The overall hardware utilization is 0.25 in the first case and 0.75 in the second case. For the spatial mapping fragmentation example each cycle executes 1024 MAC operations. However, only 256 operations contribute to the layer's result. The other 768 operations are dark operations.

approach is used since it combines well with the scaling factor approach and avoids an individual compute roof for each layer.

The next scaling factors discussed are $scale_{ifmap}$ and $scale_{ofmap}$. The former can be described as follows for the NVDLA full configuration, where $atom_{AXI}/atom_{NVDLA} = 2$, i.e. the AXI bus width is twice the size of the internal NVDLA word width:

$$scale_{ifmap} = pad(i_c, b_i) \cdot \frac{b_i}{i_c} + \frac{d_{darkBW}(i_w, i_h, i_c, b_i)}{i_c \cdot i_w \cdot i_h}$$

$$pad(c, b) = \left\lceil \frac{c \cdot b}{atom_{NVDLA}} \right\rceil \cdot b^{-1} \cdot atom_{NVDLA}$$

$$d_{darkBW}(w, h, c, b) = (w \bmod 2) \cdot h \cdot pad(c, b) \cdot b$$

Here four influences on the scaling factor explained in Subsection 2.1 occur. The first one is scaling due to multi-byte **data types**. The NVDLA uses fp16 as default which results in $b_i = 2$ and linearly scales the amount of data fetched.

Secondly, **zero-padding** occurs. The NVDLA has to work with so-called *atoms* because of its internal word width. In the case of the NVDLA full configuration, an atom must consist of 32 B in the channel direction. This is represented by the parameter $atom_{NVDLA}$. If this is not the case, zero-padding must be applied. For example, for fp16 data types the channels are always padded to be a multiple of 16. So, $i_c = 7$ is padded to 16 channels, $i_c = 17$ to 32 channels and so on.

The third influence on the scaling factor is **dark bandwidth**. Since the $atom_{NVDLA}$ is 32 B while the atom of the bus is 64 B ($atom_{AXI}$) requesting an odd number of atoms will lead to dark bandwidth. Because the NVDLA reads data row-wise, an odd row size will lead to dark bandwidth. So, for every row there are 32 B of dark bandwidth.

Lastly, **data reload** occurs. In the previous formulas it was assumed that ifmap and kernel fit into the 512 KiB convolution buffer of the NVDLA full configuration. However, if this is not the case, the ifmap will be broken into multiple tiles similar to the algorithm proposed by Zhang et. al. [6]. These

tiles have overlapping areas which result in overall increase of ifmap data. Since the NVDLA treats the individual tiles as separate layers, this should also be done in the analytical model. Otherwise, the scaling factor will quickly become complex.

The last scaling factor to be discussed for convolutional layers is the weight scaling factor $scale_{weight}$. Basically, the total number of weights is equal to the sum of the volumes of the kernel cuboids multiplied with the data type and zero-padded to be aligned with the convolutional buffer's width $cbuf_{width}$. This results in the following scaling factor:

$$scale_{weight} = \left\lceil b_k \cdot \frac{k_w \cdot k_h \cdot i_c \cdot k_n}{cbuf_{width}} \right\rceil \cdot \frac{cbuf_{width}}{k_w \cdot k_h \cdot i_c \cdot k_n}$$

Since the amount of weights is often much greater than $cbuf_{width}$ which is 128 B for the NVDLA full configuration, a scaling factor of of $scale_{weight} \approx 2$ is observed for most fp16 cases. The scaling factor for the ofmap is assumed to be 0, since convolutional layers are usually pipelined with a bias layer which will be considered in the following:

$$scale_{ofmap} = 0$$

The next layer to be considered is the bias layer. It always succeeds a convolutional layer and is executed in a pipelined fashion on the NVDLA's SDP. Since it has a fixed throughput of $throughput_X$ ifmap elements per cycle, it is straightforward to determine the operational roof and operation scaling factor as follows:

$$comp_{roof} = throughput_X \cdot clock$$

$$scale_{ops} = \left\lceil \frac{i_w \cdot i_h \cdot pad(i_c, b_i)}{throughput_X} \right\rceil \cdot \frac{throughput_X}{o_w \cdot o_h \cdot o_c \cdot k_w \cdot k_h \cdot k_c}$$

Since a bias layer is always pipelined after a convolutional or an IP layer, there is no ifmap data to fetch, so:

$$scale_{ifmap} = 0$$

The ofmap follows the same principles as before:

$$scale_{ofmap} = pad(o_c, b_o) \cdot \frac{b_o}{o_c} + \frac{d_{darkBW}(o_w, o_h, o_c, b_o)}{o_c \cdot o_w \cdot o_h}$$

As the formula shows, the case $o_w = o_h = 1$ is particularly problematic, because here about 50% of the data traffic would consist of dark bandwidth. This is the case after every fully connected layer. Therefore, the NVDLA can be operated in a *compact mode* in which data is read channel-wise rather than row-wise. This reduces the dark bandwidth to a minimum, resulting in the following formula:

$$d_{darkBW}(w, h, c, b) = atom_{NVDLA} \cdot \left(\left\lceil \frac{c \cdot b}{atom_{NVDLA}} \right\rceil \bmod 2 \right)$$

The bias data is transmitted sequentially, therefore the scaling for the weights depends only on the dark bandwidth related to the data type:

$$scale_{weight} = \left\lceil \frac{k_n \cdot b_k}{atom_{AXI}} \right\rceil \cdot atom_{AXI}$$

Since one bias value is needed per output channel, $k_w = k_h = k_c = 1$ and $k_n = o_c$ applies. In addition, a bias has no influence on the dimensions, so $i_w = o_w$, $i_h = o_h$ and $i_c = o_c$ always apply.

Besides bias and convolutional layers there are a number of other layer types for which this methodology was applied. However, these are much less performance-critical and will not be discussed in detail as this would go beyond the scope of this paper.

3.4 Tiling

As already mentioned in Subsection 2.1 the local memory capacity of many DLAs is not sufficient to store ifmap, ofmap, and weight data locally. For example, using the NVDLA's data format, the first layer of AlexNet already comprises more than 1MiB of ifmap data exceeding the convoluational buffer's capacity of 512KiB. Since data is usually used more than once for calculations, especially for convolutions, an optimal reuse strategy is crucial for fast and efficient operation of a DLA. Typical reuse strategies are the fused layer approach of Alwani et al. [3] or the tiling algorithm from Zhang et al. [6].

In the following, the reuse strategy of the NVLDA is examined in more detail. By analyzing the NVDLA's compiler we found out that a convolution can be executed in 6 different modes depending on the size of the data:

1. Full ifmap and full weight (no split needed)
2. Full ifmap and kernel groups as ping-pong
3. Full ifmap and one kernel group
4. Partial ifmap (h-tiled) and full weights
5. Partial ifmap (h-tiled) and kernel groups as ping-pong
6. Partial ifmap (h-tiled) and one kernel group

The first 2 modes represent a standard convolution where most of the data can be kept locally and thus no performance penalty is to be expected. In Mode 3 the convolution buffer can hold the whole ifmap but only one kernel group (16 kernel data cubes). This mode might come with a huge performance penalty as a parallel execution of fetching weights and executing the convolution is not guaranteed anymore. Modes 4, 5, 6 apply a reuse strategy that can be summarized as a simplified version of the tiling algorithm by Zhang et al. [6]. In these modes the ifmap is horizontally subdivided into multiple tiles as depicted in Fig. 6. This drastically reduces the amount of ifmap data that has to be kept locally. One drawback, however, is that parts of the ifmap have to be loaded several times, which is represented by the dark orange parts in Fig. 6. As the number of tiles increases, so does the relative amount of data

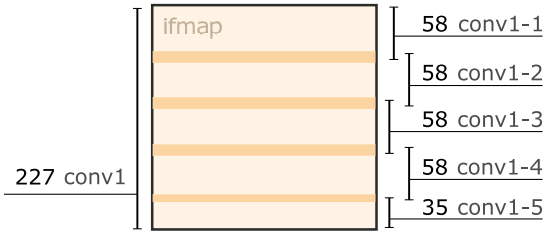

Fig. 6 The NVDLA tiling algorithm applied on AlexNet's first layer. Instead of one ifmap with 227 pixels in height direction, 5 tiles with 58 and 35 pixels respectively are convoluted.

loaded multiple times, therefore an optimal compiler should always minimise the number of tiles. Note, that this optimization problem is quite simple as the NVDLA compiler only supports horizontal tiling. According to the source code a vertical tiling and a tiling in channel direction will be introduced in the future.

To model the aspect of data overhead caused by tiling one can either adjust the corresponding scaling factors, or regard each tile as a single layer. Our model uses the latter approach. This way scaling factors are kept simple and a possible tiling directly becomes apparent from the results. Whether a layer needs tiling and what sizes these layer are was determined by a compiler mockup we derived from the original NVDLA compiler.

3.5 Results

In this subsection AMAIX parameterized for the NVDLA, as shown in the previous subsection, is used to predict inference performance for the AlexNet and LeNet CNNs. The results are compared to the inference performance measured on the hybrid prototype introduced in Subsection 3.2. The non-cycle accurate SystemC-TLM model was used for measuring the exchanged data amounts with the main memory while the cycle accurate Verilog model was used to measure the time a layer needs for its execution. In addition, the NVDLA performance estimator spreadsheet provided by Nvidia was used for comparison [2]. The standard KMD, UMD and NVDLA compiler in basic mode were used to execute the following measurements. All KMD debug output was removed for the RTL measurements, as it turned out to reduce performance significantly.

As a first example the results of LeNet shall be analyzed which are depicted in Table 1. A corresponding roofline graph can be found in Fig. 7. The following parameters are shown in the table: execution time of a layer according to AMAIX ($t_{layer,am}$), the hybrid prototype ($t_{layer,hy}$) and the NVDLA performance sheet ($t_{layer,ps}$). Furthermore, the boundary (either memory or compute bound), data exchanged with main memory (d_{weight}, d_{ifmap}, d_{ofmap}) and number of operations (n_{ops}) are also displayed. The amount of data refers to both simulation and analytical model, since the model predicted this 100% ac-

Table 1 Results of LeNet. In all cases the bias is dominated by the corresponding convolutional/fully connected layer. For a calculation example see Appendix 6.1.

Layer	$t_{layer,am}$	$t_{layer,hy}$	$t_{layer,ps}$	bound	d_{weight}	d_{ifmap}	d_{ofmap}	n_{ops}
Unit	µs	µs	µs	$\{c, m\}$	B	B	B	*operations*
conv1	28.8	28.9	7.2	c	1,024	25,088	0	29,491,200
(bias)	0	0	0	-	64	0	36,864	18,432
pool1	4.61	4.61	0	c	0	36,864	9,216	18,432
conv2	6.40	6.93	3.2	c	50,048	9,216	0	6,553,600
(bias)	0	0	0	-	128	0	8,192	4,096
pool2	1.02	1.06	0	c	0	8,192	2,048	4,096
fc3	12.5	12.97	15.67	m	800,000	2,048	0	8,388,608
(bias)	0	0	0	-	1,024	0	1,024	512
relu3	0.03	0.08	0	c/m	0	1,024	1,024	128
fc4	0.18	0.37	0.17	m	10,112	1,024	0	131,072
(bias)	0	0	0	-	64	0	64	32
softmax	0	0	0	-	0	0	0	0
idle	0	20.52	0	-	0	0	0	0
Total	53.9	54.92+20.52	26.2	-	862,464	83,712	58,432	44,610,208

curately. The number of operations refers only to the analytical model. Layers which are pipelined and not dominant are enclosed in brackets. The execution time of a layer is defined as the time between setting the kick-off register and raising the interrupt flag. The results show that for the total inference time the analytical model predicts the measured inference time of 54.9 µs with 53.9 µs or 98% accuracy. The model deviates from the measurements in some layers, especially in small layers. Several reasons for this were discovered during analysis. Firstly, the individual subprocessors such as the SDP also have internal pipelines which need a certain number of cycles to be filled. Furthermore, a certain amount of data must be available before the execution of operations can be started with. How these effects be can included in the Roofline model to improve the accuracy is shown in the next subsection.

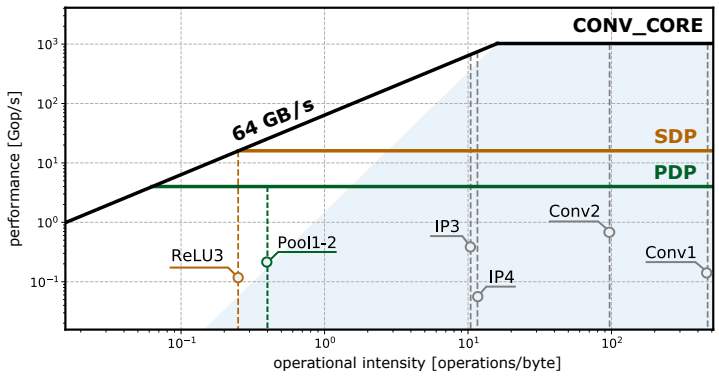

Fig. 7 Applying a roofline graph on the obtained results of LeNet.

The time obtained from the NVDLA performance sheet overestimated the performance of the NVDLA by more than 2x. This is due to the performance sheet assuming optimizations like layer pipelining or Winograd convolution which were not implemented in the NVDLA compiler at the time the performance sheet was released. Due to the lack of configurability, the result given by the performance sheet could not be improved further. Some aspects like ifmap tiling or bias layers are completely omitted, which further deteriorates its accuracy.

Fig. 8 shows an activity trace of LeNet on the hybrid prototype. The ”idle” parts represents phases in which the NVDLA waits for further instructions from the driver. These times have already been greatly reduced by the previously mentioned modifications of the KMD, but are still significant. One reason for this is that LeNet is a very small example for today's standards. As the results show, the NVDLA can process most layers within a few hundred cycles or less. This leads to the hardware being faster than the driver.

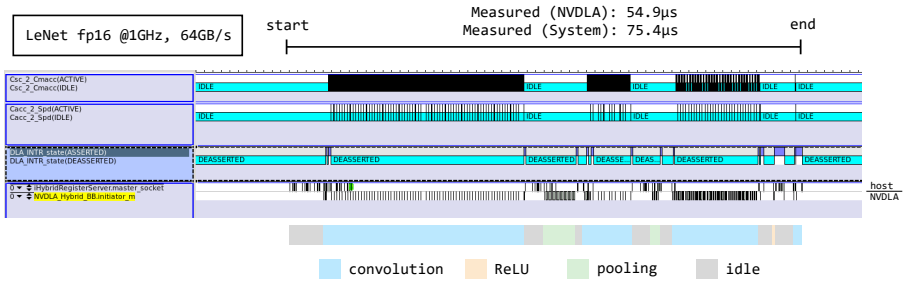

Fig. 8 Activity trace of LeNet running as a Hybrid Emulation.

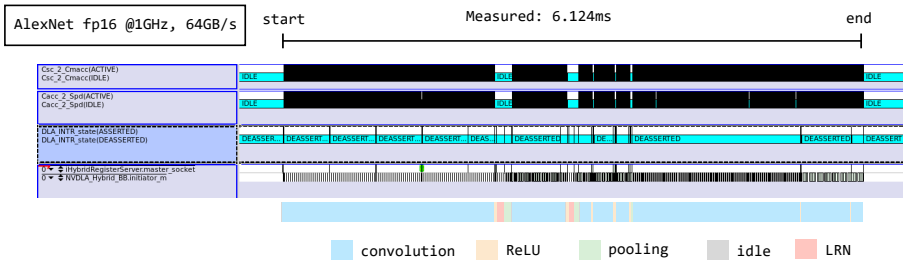

Fig. 9 Activity trace of AlexNet running as a Hybrid Emulation.

Therefore, the more recent AlexNet is analyzed in the following. As can be seen in Fig. 9, AlexNet's execution time of 6.124 ms is about 100 times longer than the execution time of LeNet. In this case, there is no idle phase observable, since the driver is now faster than the hardware. In contrast to LeNet, the NVDLA also reaches the capacity limits of its internal memory

for AlexNet. For this reason the first convolution must be split into 5 tiles (see Table 2), as the ifmap does not fit into the 512 KiB convolution buffer as a whole. Using this information from the compiler mockup the analytical model was able to predict the amount of data required 100% accurately. The total inference time of 6.124 ms was accurately predicted to about 88% with an estimate of 5.416 ms. The performance sheet overestimated the performance of the NVDLA again with 2.3 ms by a factor of 2.7x. The roofline graph depicted in Fig. 10 shows a lot of similarity to LeNet's graph. Again a huge variety of operational intensities for convolutional layers can be oberserved, reaching from 888 Ops/Byte for the first layer to 8 Ops/B for the last one.

Table 2 Results of AlexNet. In all cases the bias is dominated by the corresponding convolutional/fully connected layer.

Layer	$t_{layer,am}$	$t_{layer,hy}$	$t_{layer,ps}$	bound	d_{weight}	d_{ifmap}	d_{ofmap}	n_{ops}
Unit	µs	µs	µs	$\{c, m\}$	B	B	B	*operations*
conv1-1	479.2	497.2	102.9	c	69,760	423,168	0	490,659,840
(bias)	0	0	0	-	192	0	129,024	63,360
conv1-2	479.2	497.2	0	c	0	423,168	0	490,659,840
(bias)	0	0	0	-	192	0	129,024	63,360
conv1-3	479.2	497.2	0	c	0	423,168	0	490,659,840
(bias)	0	0	0	-	192	0	129,024	63,360
conv1-4	479.2	497.2	0	c	0	423,168	0	490,659,840
(bias)	0	0	0	-	192	0	129,024	63,360
conv1-5	279.5	279.5	0	c	0	255,360	0	286,218,240
(bias)	0	0	0	-	192	0	75,264	36,960
relu1	18.5	18.2	0	c/m	0	591,360	591,360	290,400
norm1	72.6	79.3	0	c	0	654,720	654,720	290,400
pool1	72.6	72.6	0	c	0	591,360	145,152	290,400
conv2	583.2	588	218.7	c	614,400	145,152	0	597,196,800
(bias)	0	0	0	-	512	0	387,072	186,624
relu2	12.1	11.7	0	c/m	0	387,072	387,072	186,624
norm2	46.7	50.8	0	c	0	428,544	897,536	186,624
pool2	46.7	46.7	0	c	0	387,072	93,184	186,624
conv3	146.0	149.6	64.9	c	1,769,472	93,184	0	149,520,384
(bias)	0	0	0	-	768	0	139,776	64,896
relu3	4.4	4.1	0	c/m	0	139,776	139,776	64,896
conv4	219	224.4	48.7	c	1,327,104	139,776	0	224,280,576
(bias)	0	0	0	-	768	0	139,776	64,896
relu4	4.4	4.1	0	c/m	0	139,776	139,776	64,896
conv5	146	151.4	32.4	c	884,736	139,776	0	149,520,384
(bias)	0	0	0	-	512	0	93,184	43,264
relu5	2.9	2.7	0	c/m	0	93,184	93,184	43,264
pool5	10.8	10.8	0	c	0	93,184	18,432	43,264
fc6	1180.2	1792.6	1152.4	m	75,497,472	18,432	0	603,979,776
(bias)	0	0	0	-	8,192	0	8,192	4,096
relu6	0.3	0.3	0	c/m	0	8,192	8,192	4,096
fc7	524.7	525.5	512.3	m	33,554,432	8,192	0	268,435,456
(bias)	0	0	0	-	8,192	0	8,192	4,096
relu7	0.3	0.3	0	c/m	0	8,192	8,192	4,096
fc8	128.2	128.8	125.2	m	8,192,000	8,192	0	66,060,288
(bias)	0	0	0	-	2,048	0	2,048	1,008
softmax	0	0	0	-	0	0	0	0
Total	5415.5	6124.4	2257.4	-	$121.9 \cdot 10^6$	$6 \cdot 10^6$	$4.6 \cdot 10^6$	$4.3 \cdot 10^9$

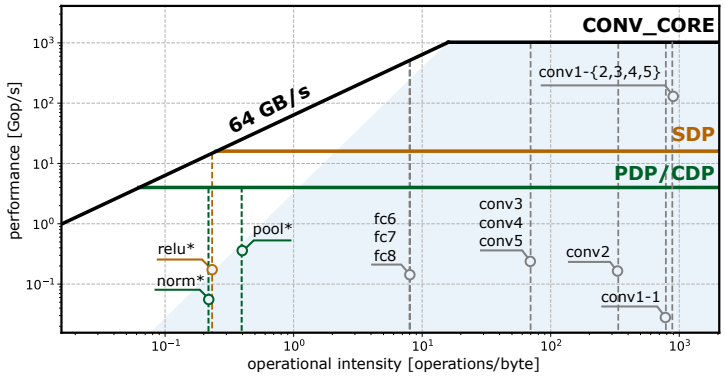

Fig. 10 Applying a roofline graph on the obtained results of AlexNet.

The largest deviation of the analytical model is found in layer "fc6". An analysis of this layer showed that the size of the required data causes the compiler to switch the convolution buffer to work in a single buffer mode, so that convolution and memory transfers no longer run in parallel. This corresponds to Mode 3 as described in Subsection 3.4. To model this effect the corresponding layer can be divided into a compute task and a memory task. This approach is pursued in the following subsection.

3.6 Divide and Conquer

As already mentioned one flaw of Roofline model is the ideal assumption of a perfect parallelism (see Fig. 11 a)). According to the Roofline model, a task comprises two subtasks: a memory subtask, and a compute subtask, both running in parallel whereby the slower task determines the overall performance. This assumption is already incorrect to some extent for any compute process, as an initial chunk of data has to be fetched before any calculations can begin. Furthermore, the result can only be written back after all calculations have been done (see Fig. 11 b)). In some cases the data subtask and compute subtask run entirely sequentially. This behaviour was observed for AlexNet's "fc6" layer. Consequently, the Roofline model, and thus AMAIX, underestimates the time required for a layer.

A more accurate model would subdivide a layer even further into different phases, and apply the roofline model for each of them. This follows the same divide-and-conquer paradigm already mentioned in the introduction where a whole CNN was split in multiple layers in order to increase the model's accuracy. Using this refined approach for the example depicted in Fig. 11 b) would result in five possible phases p_1, ..., p_5 which have to be considered accordingly.

In order to establish such a model for the NVDLA, we analyzed the source code and ran annotated SystemC simulations to obtain the results depicted in

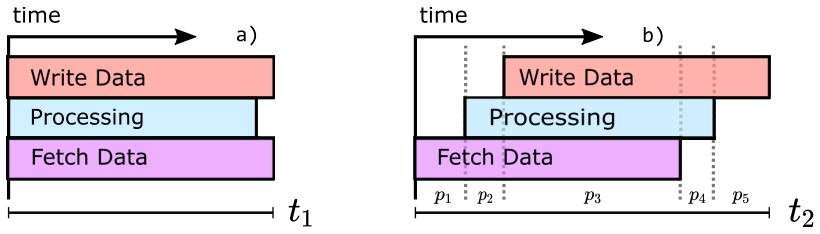

Fig. 11 Comparing the time to finish a layer from the established analytical model (t_1) to a more realistic case (t_2).

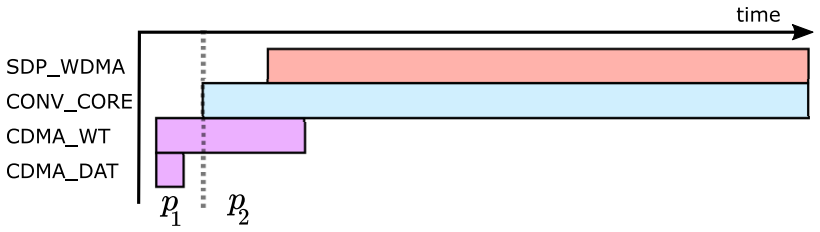

Fig. 12 Activity of different hardware units for LeNet's first convolutional layer. `CDMA_DAT` is a DMA responsible for fetching the ifmap while `CDMA_WT` fetches the weights. The processing results from `CMAC` are written back to the main memory by the `CDMA_WT`. Note that the annotated SystemC simulation is not as accurate as the RTL simulation, but it's functional correctness already allows deep insights.

Fig. 12. The investigation showed that the NVDLA can exhibit a considerable warm-up phase for convolutions. In order to start with the processing, the whole ifmap needs to be fetched and at least one so-called kernel group (a group of 16 kernel cubes) needs to be available. In order to keep the model simple, the convolution is only divided into these two most important phases. As can be seen in Fig. 12, further phases can easily be defined if desired. A mathematical description can be formulated as follows:

$$t_{p_1} = \frac{d_{p_1}}{memory_{peak}}$$

$$d_{p_1} = \begin{cases} d_{kgroup} + d_{ifmap} & \text{if } d_{kgroup} > d_{ifmap} \\ d_{ifmap} + min(d_{ifmap}, d_{weight}) & \text{if } d_{kgroup} \leq d_{ifmap} \end{cases}$$

$$d_{kgroup} = \left\lceil b_k \cdot \frac{min(16, k) \cdot k_h \cdot k_w \cdot i_c}{atom_{AXI}} \right\rceil \cdot atom_{AXI}$$

The second phase still follows the equations formulated in Section 2.1. Only the amount of data needs to be adjusted for the initial chunk that will already be loaded:

$$d_{total} = d_{ifmap} + d_{weight} + d_{ofmap} - d_{p_1}$$

The model can be further improved by considering layers that need to run in a sequential fashion due to convolution buffer capacity constraint (Mode

6, Subsection 3.4). For example, assuming a sequential execution instead of parallelized one increases AlexNet's layer "fc6" execution time from 1180.2µs to 1769.8µs. While modeling an offset phase only increased the accuracy by about 1% per layer, assuming a sequential execution had by far the biggest impact. The analytical model's accuracy increased from 88% to 98% for AlexNet.

The drawback of the divide-and-conquer approach is an increased complexity of the analytical model. In case of the NVDLA, the new model also introducess phases with operational intensities of 0 or infinity, making it difficult to plot them in roofline graphs.

3.7 Design Space Exploration

Since the simulation results have shown that AMAIX allows precise predictions to be made, it will be used to explore the NVDLA's design space in this section. The NVDLA has over 30 different hardware parameters that can be individually set to provide a suitable configuration for each application. Some of these parameters are also found in the analytical model. For example, the width and depth (T_c (depth) and T_k (width)) of `CONV_CORE`'s MAC unit. The performance of AlexNet in frames per second is shown regarding these parameters in Fig. 13. The tuples (width, depth) are chosen such that their product is constant, which corresponds to a constant area. The analysis shows that besides the full configuration of the NVDLA there are other configurations that theoretically allow a faster execution of AlexNet. However, with a convolution buffer size of 512 KiB the design space is limited to the highlighted area of the graphs. Thus the NVDLA full configuration seems to be optimal for AlexNet given the convolution buffer constraint. This result cannot be verified using

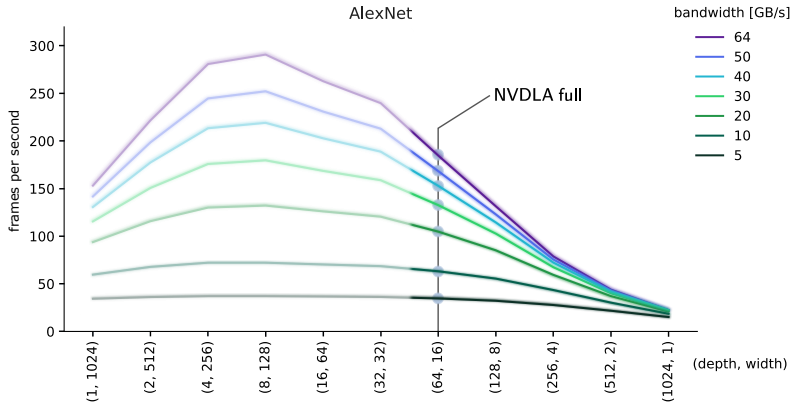

Fig. 13 Design space exploration of the `CONV_CORE` using AlexNet.

the NVDLA. Although the hardware synthesis of the NVDLA configurations

is possible, the drivers so far only support the full, large and small NVDLA configuration.

4 Conclusion & Outlook

In this paper the novel AMAIX aproach for the inference performance estimation of DLAs was proposed and evaluated. AMAIX's design allows for a generic representation of DLAs, due to its configurable scaling factors. Its per layer modeling approach is a reasonable compromise between model complexity and accuracy as shown in the detailed case study. In case the accuracy is considered not high enough, it was shown that layers can be further split up using a divide-and-conquer paradigm, to which the Roofline model is then applied again. As shown in the conducted case-study using the NVDLA, the layer-wise AIMAX model predicted the inference time with an accuracy of 88% for AlexNet and 98% for LeNet compared to an accurate RTL emulation. For AlexNet the accuracy can be increased to 98% by using the more detailed model. In addition, it was shown that AMAIX can be used for design space exploration, especially since it can be evaluated several orders of magnitude faster than a Verilog or SystemC simulation.

In future work, it would be interesting to apply AMAIX to other DLA architectures as well. Another promising application are compiler optimizations. At many points, a compiler for DLAs must make the decision whether to accept additional data transfers for more compute performance. This concerns, for example, the selection of the convolution mode from Section 3.4. Here, the compiler has to decide whether the ifmap is tiled, i.e. more data transfers are generated, or whether the single buffer mode is used, which reduces performance due to its sequential execution. Using an analytical model, a sophisticated decision could be made at this point. A further example is the decision between Winograd convolution and standard convolution. While Winograd convolution can usually be calculated much faster, it still requires more weight data. It should therefore only be used if the system is compute bound. This compiler could be further refined to a situation-aware just-in-time compiler as some parameters such as the available memory bandwidth are hard to predict in advance.

5 Acknowledgements

This work was supported by Synopsys GmbH.

References

1. LeNet Prototxt. github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt. Accessed: 27.08.2021
2. NVDLA Github Repository. https://github.com/nvdla. Accessed: 27.07.2019
3. Alwani, M., Chen, H., Ferdman, M., Milder, P.: Fused-Layer CNN Accelerators. 49th IEEE/ACM International Symposium on Microarchitecture (MICRO) (2016)
4. Bratt, I.: Arm's First-Generation Machine Learning Processor. IEEE Hot Chips 30 Symposium (2018)
5. Chen, Y., Emer, J.S., Sze, V.: Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks. CoRR (2018)
6. Chen Zhang Peng Li, G.S.: Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (2015)
7. Hill, M., Janapa Reddi, V.: Gables: A Roofline Model for Mobile SoCs. In: IEEE Int. Symposium on High Performance Computer Architecture (HPCA) (2019)
8. Jünger, L., Zurstrassen, N., Kogel, T., Keding, H., Leupers, R.: Amaix: A generic analytical model for deep learning accelerators. In: SAMOS International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 36–51. Springer (2020)
9. Kogel, T.: Synopsys Virtual Prototyping for Software Development and Early Architecture Analysis, pp. 1127–1159. Springer Netherlands (2017)
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems (2012)
11. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, VOL. 86 (1998)
12. Misko, J., Jadhav, S.S., Kim, Y.: Extensible embedded processor for convolutional neural networks. Scientific Programming **2021** (2021)
13. Norman P. Jouppi Cliff Young, N.P.: In-Datacenter Performance of a Tensor Processing Unit. 44th Int. Symposium on Computer Architecture (ISCA) (2017)
14. Reagen, B., Whatmough, P., Adolf, R., Rama, S., Lee, H., Lee, S.K., Hernández-Lobato, J.M., Wei, G.Y., Brooks, D.: Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA) (2016)
15. Venkataramanan, G.: Compute and Redundancy Solution for the Full Self-Driving Computer. IEEE Hot Chips 31 Symposium (2019)
16. Williams, S., Waterman, A., Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM (2009)

6 Appendix

6.1 Example Calculations for LeNet

The following calculations of LeNet's *conv1* and subsequent *bias* layer showcase how the results of Table 1 are obtained. Layer and kernel sizes from were taken from *Caffe's* publicly accessible LeNet implementation [1].

6.1.1 NVDLA configuration and layer sizes

$$T_c = 64,\ T_k = 16,\ cbuf_{width} = 128,\ throughput_X = 16,$$
$$b_i = b_k = b_o = 2$$
$$memory_{peak} = 64GB/s,\ clock = 1GHz$$

$$conv1 : i = (28, 28, 1),\ k = (5, 5, 1, 20),\ o = (24, 24, 20)$$
$$bias : i = (24, 24, 20),\ k = (1, 1, 1, 20),\ o = (24, 24, 20)$$

6.1.2 Number of ifmap bytes for conv1

$$\begin{aligned} pad(c, b) &= \left\lceil \frac{c \cdot b}{atom_{NVDLA}} \right\rceil \cdot b^{-1} \cdot atom_{NVDLA} \\ pad(1, 2) &= \left\lceil \frac{1 \cdot 2}{32} \right\rceil \cdot 0.5 \cdot 32 \\ &= 16 \end{aligned}$$

$$\begin{aligned} d_{darkBW}(w, h, c, b) &= (w \bmod 2) \cdot h \cdot pad(c, b) \cdot b \\ d_{darkBW}(28, 28, 1, 2) &= (28 \bmod 2) \cdot 28 \cdot pad(1, 2) \cdot 2 \\ &= 0 \end{aligned}$$

$$\begin{aligned} scale_{ifmap} &= pad(i_c, b_i) \cdot \frac{b_i}{i_c} + \frac{d_{darkBW}(i_w, i_h, i_c, b_i)}{i_w \cdot i_h \cdot i_c} \\ &= pad(1, 2) \cdot \frac{2}{1} + \frac{d_{darkBW}(28, 28, 1, 2)}{1 \cdot 28 \cdot 28} \\ &= 32 \end{aligned}$$

$$\begin{aligned} d_{ifmap} &= scale_{ifmap} \cdot i_w \cdot i_h \cdot i_c \\ &= 32 \cdot 28 \cdot 28 \cdot 1 \\ &= 25,088 \end{aligned}$$

6.1.3 Number of weight bytes for conv1

$$\begin{aligned} scale_{weight} &= \left\lceil b_k \cdot \frac{k_w \cdot k_h \cdot i_c \cdot k_n}{cbuf_{width}} \right\rceil \cdot \frac{cbuf_{width}}{k_w \cdot k_h \cdot i_c \cdot k_n} \\ &= \left\lceil 2 \cdot \frac{5 \cdot 5 \cdot 1 \cdot 20}{128} \right\rceil \cdot \frac{128}{5 \cdot 5 \cdot 1 \cdot 20} \\ &= 2.048 \end{aligned}$$

$$\begin{aligned} d_{weight} &= scale_{weight} \cdot k_w \cdot k_h \cdot k_c \cdot k_n \\ &= 2.048 \cdot 5 \cdot 5 \cdot 1 \cdot 20 \\ &= 1024 \end{aligned}$$

6.1.4 Number of operations for conv1

$$\begin{aligned} scale_{ops} &= \left\lceil \frac{i_c}{T_c} \right\rceil \cdot \left\lceil \frac{k_n}{T_k} \right\rceil \cdot \frac{T_k \cdot T_c}{i_c \cdot k_n} \\ &= \left\lceil \frac{1}{64} \right\rceil \cdot \left\lceil \frac{20}{16} \right\rceil \cdot \frac{16 \cdot 64}{1 \cdot 20} \\ &= 102.4 \end{aligned}$$

$$\begin{aligned} n_{ops} &= scale_{ops} \cdot o_w \cdot o_h \cdot o_c \cdot k_w \cdot k_h \cdot k_c \\ &= 102.4 \cdot 24 \cdot 24 \cdot 20 \cdot 5 \cdot 5 \cdot 1 \\ &= 29,491,200 \end{aligned}$$

6.1.5 Number of ofmap bytes bias

$$\begin{aligned} pad(20,2) &= \left\lceil \frac{20 \cdot 2}{32} \right\rceil \cdot 0.5 \cdot 32 \\ &= 32 \end{aligned}$$

$$\begin{aligned} d_{darkBW}(24,24,20,2) &= (24 \, mod \, 2) \cdot 24 \cdot pad(20,2) \cdot 2 \\ &= 0 \end{aligned}$$

$$\begin{aligned} scale_{ofmap} &= pad(o_c, b_o) \cdot \frac{b_o}{o_c} + \frac{d_{darkBW}(o_w, o_h, o_c, b_o)}{o_c \cdot o_w \cdot o_h} \\ &= pad(20,2) \cdot \frac{2}{20} + \frac{d_{darkBW}(24,24,20,2)}{20 \cdot 24 \cdot 24} \\ &= 3.2 \end{aligned}$$

$$\begin{aligned} d_{ofmap} &= scale_{ofmap} \cdot o_w \cdot o_h \cdot o_c \\ &= 3.2 \cdot 24 \cdot 24 \cdot 20 \\ &= 36,864 \end{aligned}$$

6.1.6 Number of operations for bias

$$\begin{aligned} scale_{ops} &= \left\lceil \frac{i_w \cdot i_h \cdot pad(i_c, b_i)}{throughput_X} \right\rceil \cdot \frac{throughput_X}{o_w \cdot o_h \cdot o_c \cdot k_w \cdot k_h \cdot k_c} \\ &= \left\lceil \frac{24 \cdot 24 \cdot pad(20,2)}{16} \right\rceil \cdot \frac{16}{24 \cdot 24 \cdot 20 \cdot 1 \cdot 1 \cdot 1} \\ &= 1.6 \end{aligned}$$

$$\begin{aligned} n_{ops} &= scale_{ops} \cdot o_w \cdot o_h \cdot o_c \cdot k_w \cdot k_h \cdot k_c \\ &= 1.6 \cdot 24 \cdot 24 \cdot 20 \cdot 1 \cdot 1 \cdot 1 \\ &= 18,432 \end{aligned}$$

6.1.7 Performance, total data and layer time

$$\begin{aligned} d_{total} &= d_{ifmap} + d_{weight} + d_{ofmap} \\ &= 25,088 + 1024 + 36,864 \\ &= 62,976 \end{aligned}$$

$$\begin{aligned} performance_{peak}(conv1) &= T_k \cdot T_c \cdot clock \\ &= 16 \cdot 64 \cdot 1GHz \\ &= 1024Gop/s \end{aligned}$$

$$\begin{aligned} op_{intensity}(conv1) &= n_{ops}/d_{total} \\ &= 29,491,200/62,976 \\ &= 468.29 \end{aligned}$$

$$\begin{aligned} performance(conv1) &= \min(\ 1024Gop/s,\ 4.68 \cdot 64Gop/s\ ) \\ &= 1024Gop/s \end{aligned}$$

$$\begin{aligned} performance_{peak}(bias) &= throughput_X \cdot clock = 16 \cdot 1GHz \\ &= 16Gop/s \end{aligned}$$

$$\begin{aligned} op_{intensity}(bias) &= 18,432/62,976 \\ &= 0.293 \end{aligned}$$

$$\begin{aligned} performance(bias) &= \min(\ 16Gop/s,\ 0.293 \cdot 64Gop/s\ ) \\ &= 16Gop/s \end{aligned}$$

$$\begin{aligned} t_{layer} &= \max\left(\frac{29,491,200}{1024 \cdot 10^9}s,\ \frac{18,432}{16 \cdot 10^9}s\right) \\ &= \max\left(28.8\mu s,\ 1.15\mu s\right) \\ &= 28.8\mu s \end{aligned}$$