SimPyler: A Compiler-based Simulation Framework for Machine Learning Accelerators

Yannick Braatz*, Dennis Sebastian Rieber*, Taha Soliman*, and Oliver Bringmann†
{yannick.braatz, dennissebastian.rieber, taha.soliman}@de.bosch.com oliver.bringmann@uni-tuebingen.de
*Robert Bosch Corporate Research, Renningen, Germany
†Eberhard Karls Universität Tübingen, Tübingen, Germany

***Abstract*—Co-optimization of hardware and software in modern deep neural network (DNN) systems can be performed using design space exploration (DSE) tools. Leveraging estimation models to predict design decisions' impact on the system's performance allows a fast evaluation of up to billions of architectural choices. In this work, we propose SimPyler, an end-to-end framework for latency estimations of DNN workloads on machine learning (ML) accelerators. SimPyler represents DNN kernels as graphs executed on abstract accelerator models to simulate the system's latency. By generating the entire simulation infrastructure automated from the DNN operator description, the framework can flexibly adjust to changes at the hardware or algorithmic level, enabling the usage in DSE applications. A key enabler in this automation process is a machine-learning compiler. The framework is implemented in Python, using only open-source software. We demonstrate and validate the proposed methodology by modeling different single-core and multi-core hardware architectures and DNNs, comparable to state-of-the-art. Our experiments show we can estimate the end-to-end latency with an average error of 4.12%**

***Index Terms*—Machine Learning Compiler, Performance Estimations, Virtual Prototypes, End-to-end**

I. Introduction

In recent years deep neural networks (DNNs) have achieved outstanding performances in solving tasks like image classification [11] and natural language processing [5].
Due to the growing complexity of emerging networks, general-purpose hardware, such as CPU or GPU, can no longer fulfill the increasing computation and memory requirements. Various domain-specific machine learning (ML) hardware accelerators have therefore been proposed to increase throughput and energy efficiency for deploying these DNNs onto embedded devices [2]–[4], [7]. Such systems generally contain similar architectural components, as depicted in Figure 1, that can highly differ in their low-level implementations. An array of processing elements (PEs) in the accelerator's core accounts for the parallel calculation of multiply and accumulate (MAC), the dominating computation for most DNNs. Data to the PE-Array in this example is supplied via a multi-level memory hierarchy, visualized by the colored regions in Figure 1. Data in this example is transferred from off-chip DRAM to the on-chip memory via direct memory access (DMA). At the same time, there can be significant variations between accelerator technologies. From the algorithmic perspective, throughput is affected by unrolling specific loops of a DNN operator along the PE-Array dimensions, determining the degree of parallel PE activity and often referred to as *spatial mapping*. Reordering and tiling of the remaining loops are called *temporal mapping* and directly contribute to the power consumption, as they dictate the re-use possibilities of data at different levels of the memory hierarchy. The combination of both is often referred to as *dataflow*. Developing high-throughput/low-energy ML accelerators demands the co-optimization of the hardware architecture along with its dataflow.

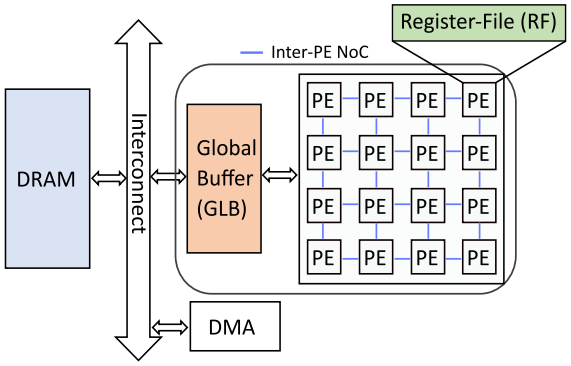

Fig. 1: A general machine learning system

However, multiple ways exist to describe a particular dataflow by modifying the DNN operator's loop-nest and various possible target hardware implementations. This leads to ample space for design decisions at the SW and HW level. Design space exploration (DSE) tools perform the dataflow/architecture co-optimization in an automated and systematical fashion under limited edge-device resources. The search efficiency is boosted by pruning the design space. Prediction models evaluate remaining design points by estimating the system's performance based on the current dataflow and hardware parameters.
As the accuracy of these models is crucial for the entire optimization process, a variety of prediction algorithms with different focuses has recently been reported [6], [10], [12], [13], [15]. In this work, we propose SimPyler (from **Sim**ulation, **Py**thon and compi**ler**), a simulation-based framework to estimate the performance of convolutional neural network (CNN) operators on ML accelerators. We can flexibly adjust to the HW and SW level modifications by automatically generating simulation models from dataflow-modified CNN operators. Moreover, combining simulation with analytical aspects in our modeling approach makes predictions accurate and comparably fast. In this paper, we make the following contributions:

DOI 10.1109/ASAP57973.2023.00042

- SimPyler provides a novel simulation-based end-to-end methodology to estimate the latency of CNNs on machine learning accelerator architectures, combining analytical and simulational attributes inside a unified model.
- By automating the process of generating simulation models for a variety of dataflows and architectures, SimPyler enables the usage in DSE applications.
- We demonstrate our methodology across multiple ML accelerator architectures and different CNN layers, showing SimPyler's ability to estimate accurate latency values for various CNN systems.

II. Background & Related Work

A. Latency Estimation Framework

Analytical approaches are favored candidates for estimation models in the context of DSE tools, as they are fast and accurate under specific assumptions. dMazeRunner [6] provides the classical 7D loop nest of CNN operators for each level of the memory hierarchy individually. Latency is computed by combining analytical models to describe hardware tasks with the corresponding CNN operator loops contributing to the task's execution time. However, it is assumed that all datatypes (inputs, filters, outputs) share the same memory instances, which is not always true in modern accelerators.

Timeloop [13], and ZigZag [12] offer a more flexible description of architecture and dataflow. Timeloop provides a template-based language to describe a design's memory and compute hierarchy along with the dataflow and mapping, using a loop-based description. Both are evaluated analytically for latency and energy estimations. However, the model might provide low prediction accuracy for conflicts on the hardware level while it demands users to learn the tool-specific language. ZigZag enables associating loops-nests to various memory levels and provides a relevance principle for each loop to ensure its reuse potential for hardware operation at that level. Though being fine-grained along the on-chip hierarchy, they do not take off-chip transfer that much into account, which can cause inaccuracies if DMA or Bus concurrency impacts the transfer latency.

Only a few simulation-based approaches have been published yet. SMAUG [15] aims to register the overhead of deploying CNNs onto ML accelerators, as configuration from a housekeeping processor, by simulating the end-to-end process. They use repeated patterns in CNN kernels to speed up their simulation. However, they do not take architectural variations in the focus of their evaluation, which is crucial in common DSEs. The authors in [10] propose a simulation methodology that leverages configurable SystemC models for the accelerator architecture. Software graphs are created from CNN kernels and executed on the abstract hardware model. However, as the simulation offers more process detail than analytical approaches, it also suffers from longer execution times. Both approaches achieve precise latency estimations as they provide a level of process detail beyond the scope of analytical models. Though, they have limited usability for DSE due to inflexibility in adjusting to changes at the CNN operator and the accelerator architecture. This is mainly because they are targeting fixed hardware. Configuring the simulation for new architectures requires a noticeable manual setup, which is infeasible for the automated exploration of large design spaces.

B. TVM

TVM [1] is an open-source machine learning compiler framework to deploy DNNs onto various general-purpose hardware and customized accelerators. Imported network models from common DNN frameworks are internally represented as hardware-agnostic intermediate representations (IRs) at the graph and operator levels. Providing API to perform modifications at both levels allows for generating optimized code for several hardware targets. At the graph level, this involves, e.g., fusing multiple operators. In contrast, the loop program of resulting operators can be modified by applying common techniques such as tiling, unrolling, reordering, or replacing specific loops with user-optimized hardware kernels. TVM ensures functional correctness for all modified programs.

C. Petri-Nets

Petri-Nets provide a simple and easy-to-implement principle to model the temporal and logical steps for executing a discrete system [16]. Directed graphs represent dependencies between process steps. The concept includes two kinds of nodes connected by the graph's edges. *Places*, that can store communication objects, called tokens, and *transitions* that can consume or produce tokens. *Token* objects are transported along the graph structure to activate nodes in the designated topological order. Transitions represent process nodes that can consume tokens to be activated and produce tokens after terminating from/to adjacent places. Places at the input and output of transitions can store a certain amount of token objects. The availability of tokens at the input determines whether a transition can be activated, while the store capabilities at the output place dictate if a transition can terminate. Each transition can consume or produce multiple tokens at once. Dependencies between process steps are represented by the graph topology, the sizes of places, and the number of tokens consumed/produced at each transition.

III. SimPyler Overview

There are three major goals a simulation-based performance estimator must achieve to find usage in DSE applications. 1) The simulation must consider the accelerator's dataflow, meaning it needs to contain optimized scheduling of the CNN operators' loop-nest to maximize the hardware architecture capabilities. 2) The accelerator hardware needs to be modeled to be precise for dynamic hardware interactions while being fast enough to allow the evaluation of many design points. 3) Different simulations must be configured without manual intervention to allow an automated search space evaluation for tuneable parameters at HW and SW levels.

An overview of the general SimPyler flow is depicted in Figure 2, wherein we achieve the aforementioned goals from the hardware and software perspective. Goal 2) is addressed by

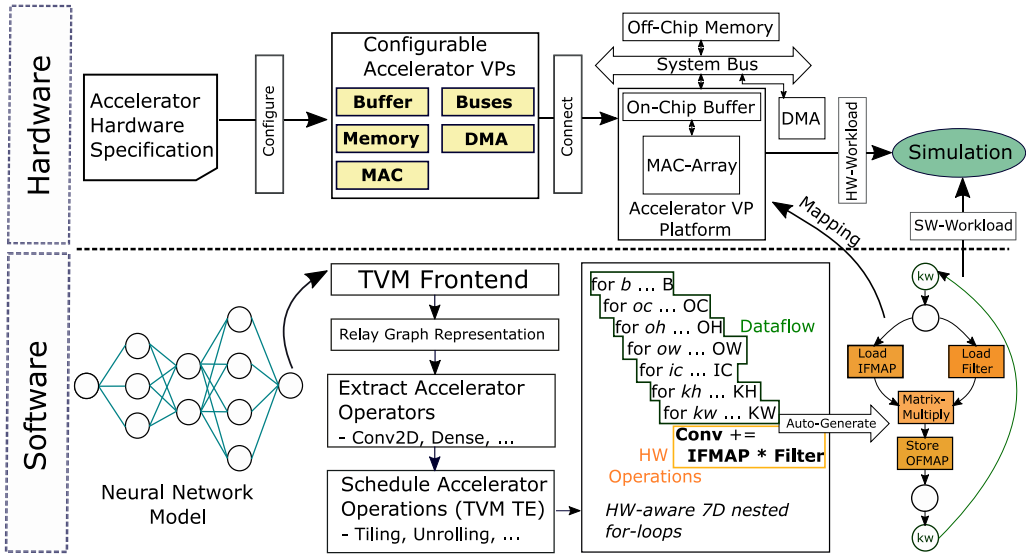

Fig. 2: SimPyler overview, combining the hardware and software perspective for latency estimations

incorporating virtual prototypes (top path) with a varying degree of process detail for representing multiple hardware architectures. 1) and 3) are achieved from the software perspective (bottom path), where TVM is utilized. We schedule CNN operator loop-nests towards dataflow representations using the TVM infrastructure. Then, for each scheduled operator, dataflow graphs are drawn that take the previously applied modifications and hardware operations to execute them into account. We gain precise latency values for the combined HW/SW system by mapping and running such graphs onto abstract accelerator models. Herein, drawing the graph from the CNN operator and performing the mapping onto accelerator architectures is entirely automated. In the following sections, we provide more details on the hardware modeling, the dataflow graph capabilities, and the automated generation and mapping of the graph and accelerator model.

IV. Hardware Perspective: VP-based Modelling of ML Accelerator Architectures

For most ML accelerators, the structural components that define the hardware layout are variations of the ones depicted in Figure 1. We observe that the isolated latencies can be well described analytically for most of these components. E.g., PE-Array structures are generally modeled in a highly deterministic fashion. Latency for performing multiply and accumulate operations of CNN layers can therefore be expressed by mathematical equations that consider the array dimensions (height and width in 2D arrays) and workload parameters as shapes and datatype of the input and output matrices. The same is true for DMA transfers if there is no traffic on the bus and sufficient storage on the destination side.

However, if the accelerator latency becomes more dependent on the interactions of its architectural components, a purely analytical description is no longer precise. In [9], the authors point out the importance of correctly considering bus bandwidth for multi-core accelerators, as it will be divided along multiple parallel bus masters and directly impacts the data transfer latency. Even for single-core accelerators, multiple DMAs can access the same bus concurrently. The bus bandwidth limits the effective DMA transfer bandwidth, which can make the whole transfer consume additional cycles. Computationally-bound systems can transform into memory-bound ones under those circumstances, making the latency challenging to estimate if the wrong boundaries are assumed. On-chip memory sizes can also violate the determinism of the entire accelerator behavior. For example, small buffers can deny a parallel supply of new buffer data while the existing data is used by the PE-Array, as data first has to be used inside the computational core before new data can be stored. Such a scenario adds stall cycles to the supplying DMA and the core. It also denies the processes dominating the accelerator functionality, namely loading inputs, calculating outputs, and storing outputs, from overlapping and running completely pipelined. Such behavior is intended in most accelerators and assumed in many analytical models, making estimations less precise if these clear pipelines no longer exist.

We use these insights by modeling ML accelerator architectures from discrete blocks that can be connected toward complete platforms. We refer to these abstract blocks in the following as virtual prototypes (VPs). Each VP embeds an analytical model, calculating the isolated VP latency for executing a certain hardware operation by including its hardware parameters and the attributes of the operation it executes. In the case of an off-chip DMA transfer, this can be something like the DMA bandwidth and the data transfer length.

As previously highlighted, several effects appear when accelerator components interact that can not easily be calculated analytically. Interactions between connected VPs are therefore simulated in an event-based flow, making each VP execution depend on the state of its neighbors. Due to dynamic inter-

hardware effects, additional clock cycles can be registered and added to the isolated block latency. If, e.g. multiple DMAs access the same bus concurrently, the bus registers events for all of them and can decide how to proceed, depending on how the bus VP is modeled for a certain accelerator. Dividing the available bandwidth between these, or applying priority to the incoming requests that make transfers of low-priority block/stall, are possible bus responses. Similarly, incoming DMA transfers can be recognized by buffers before data is loaded/stored. Depending on the transfer length and the available memory space, they can block or allow the transfer. Combining analytical and simulation capabilities makes our modeling precise, even for dynamic hardware interactions.

V. Representing Accelerator Dataflow

As the relation between dataflow and architecture is important for the accelerator's performance, we consider the same correlation for our simulations. Similar to how dataflow makes use of the accelerator's architecture, we define dataflow graphs that orchestrate the activation of blocks inside VP platforms. If we consider the VP architecture to be the backend of our simulation model, the dataflow graph provides the frontend that drives it. We have chosen a graph representation to model dataflow, because of the following reasons: 1) Graph topology can provide a systematical structure for operations that must be performed on hardware to execute a DNN operator properly. 2) Concurrent tasks at the hardware level can be represented by, e.g., parallel paths. The same applies to pipelines or repeated tasks, as we will explain further. 3) Having the hardware backend as an abstract VP model, we can directly concatenate graph nodes to VP blocks to execute them in the order the graph topology dictates.

A. Transforming CNN operators into dataflow graphs

Unoptimized CNN operators are commonly described as 7D perfectly nested loops, as depicted in Figure 3. Popular techniques can be leveraged to modify the loop structure. These include *loop blocking/tiling*, *loop reordering*, and *loop unrolling*, whereas all of them are generally applied at once for a single operator's modification. While from a pure algorithmic perspective, these changes do not affect the functional correctness of the program, they have a huge impact if executed on hardware. Blocking refers to the limited on-chip memory sizes and can split loop operations into smaller tiles that fit into these memory spaces, which adds additional dimensions to the loop nest. Reordering enables data reuse along the memory hierarchy, as datatypes indices (IFMAP, Filter, OFMAP) might be independent of the surrounding loop levels and so do not demand a reload in the next loop iteration. Unrolling is leveraged to parallelize computations on the computational core of the hardware, e.g., the PE-Array in Figure 3. All three contribute to the accelerator's dataflow and must be applied before we transform operators into dataflow graphs.

We leverage the ML compiler framework TVM to perform the dataflow-oriented operator modifications, which TVM refers to as scheduling. We use the same term in the following

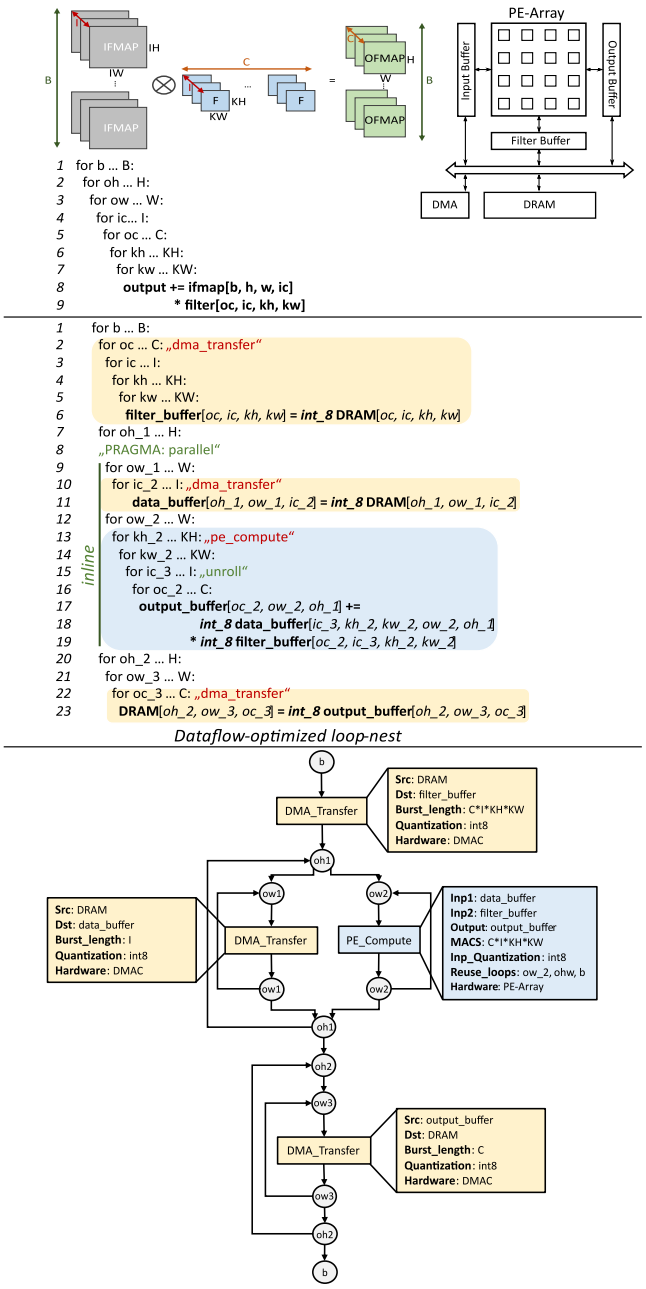

Fig. 3: (Top) CNN operator's nested loops and example accelerator; (Middle) alter the operator's loop nest for optimized scheduling on the example accelerator, using TVM's language; (Bottom) The resulting dataflow graph built by SimPyler

to describe changes applied to the loop-nest. TVM allows us to import complete DNNs and filter for operators we want to execute on our accelerator model. Using TVM's libraries enables us to apply the techniques as mentioned above to selected operators while the schedule's correctness is guaranteed.

Equally important is that we can position additional com-

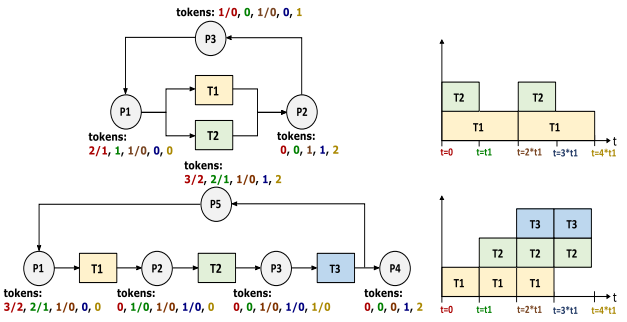

Fig. 4: Petri-Net modeling of (Top) parallel transitions and (Bottom) pipelined transitions; colors intend the tokens available at the places at different timings, wherein "/" means that a token is consumed at the same time

putations inside loop-nests and mark loops by *PRAGMAS*, which we will extensively use. As we want to automate the transformation of schedules into dataflow graphs, which is part of the following section, we need to define some schedule patterns that can always be interpreted concisely. Identifying loops by PRAGMAS plays an important role herein. Table I lists the identifiers used by SimPyler and assigned identifications. We assume that at the core of each PRAGMA loop, there is

PRAGMA name	Interpretation
dma-transfer	Transform all nested loops into DMA Burst transfer
pe-compute	Execute all nested loops onto a single PE-Array
parallel	Execute all following inline loops in parallel, on different VPs
unroll	Execute all following inline loops in parallel, on same VPs
pipeline	Execute all following inline loops in a pipeline

TABLE I: Pragmas used by SimPyler to identify graph pattern from scheduled CNN operators

a computation that can be performed on a particular block of the accelerator architecture. In Figure 3, we show how to schedule a CNN operator on the accelerator, depicted in the middle of Figure 3, and the corresponding dataflow graph below. We added computations for data transfer to/from the on-chip buffers to the loop-nest (lines 6, 11, 23 in the middle of Figure 3). Loop-nests marked by *dma-transfer* refer to the designated burst length, while the entire loop nest will be transformed into a single DMA node in the dataflow graph at the bottom of Figure 3. Note that these nodes are mapped to the DMA VP and are aware of the transfer source and destination, which are part of the VP model. Backward paths refer to the repeated node activation for the unmarked loops surrounding the transformed ones. Similar to the data transfer, we marked the computational load for the PE-Array by the *pe-compute* PRAGMA. All nested loops will be used for one MAC iteration on the PE-Array VP, while the two innermost loops are unrolled along the PE-Array's dimensions (see Pragma *unroll*). Note that we also register the above loop levels that contribute to the data reuse inside the PE-Array. We will update these loops when iterating through them along the graph, also providing this information to the Array that can leverage this information for input/output data reuse.

Following this design methodology, we can represent different dataflows at the operator level while there is a concise translation into dataflow graphs. The mapping of graph nodes to VPs of a pre-defined accelerator model is done simultaneously. Following the graph topology, each dataflow graph can drive simulation on the VP model. VPs are activated in the proposed order, while their interactions are simulated at the VP level.

B. Expressing Hardware concurrency by Petri-Nets

For each dataflow graph, we aim to perform simulation by traversing it along the graph edges, representing the flow of data in the actual hardware. Nodes herein refer to operations that can be executed on the accelerator VP model. The graph topology can express concurrent VP interactions. We leverage Petri-Nets for the graph implementation to ensure that our simulations follow the graph order and activate VPs as intended. They have proven to be an effective methodology to model digital hardware [16]. Figure 4 shows how concurrent VP behavior can be realized if the nodes and edges are implemented by *transitions* and *places*. We assume that all transitions refer to operations that VPs can perform. At the top of Figure 4, both transitions, T1 and T2, are processed in parallel. They share the same input node, meaning both are activated simultaneously, as two tokens are available at their shared input node. The timing diagram shows they start contemporaneously and synchronize at the shared output node. A token is put on the backward path after both are finished, enabling restarting of the procedure. At the bottom of Figure 4, pipelined execution of three VP operations, T1, T2, and T3, are expressed. At the start (t=0), three tokens are available in the input and backward place. T1 consumes one token from each and executes in time t1. After t=t1, T1 can start again due to the availability of new inputs, and T2 as well, as T1 has produced the necessary token on P2. Assuming both run for the time t1, at t=2*t1, T1, T2, and T3 are activated at once, while T3 produces one new token on the backward place P5 at t=3*t1 to enable a new pipeline.

These are common concurrent execution schemes observed in real hardware processes. By combining multiple individual graphs like these, we can express complex system behavior by Petri-Nets. In order to use such graphs for full simulations, we put one initial token at the start node of the graph and wait until the output node is reached.

VI. Towards Automated Performance Modelling

It is crucially important for simulations to automate the initialization if used for DSE, as they have to be reconfigured for each point in the design space. TVM represents each scheduled operator internally as an abstract syntax tree (AST) and offers API to traverse it systematically. The AST for the example from Figure 3 is simplified in Figure 5. The outermost inline

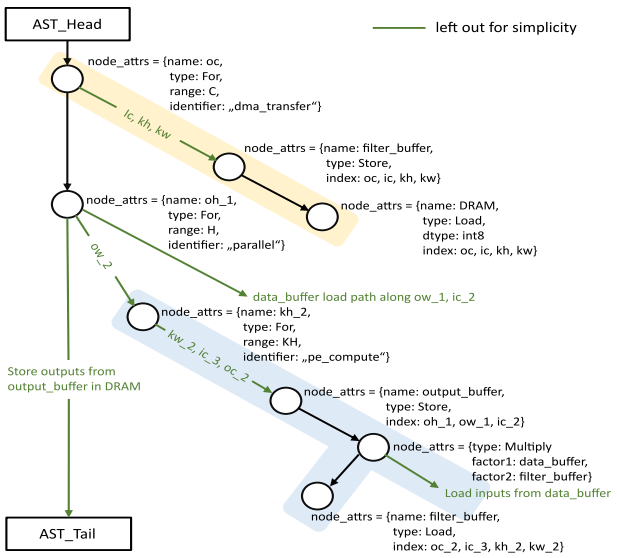

Fig. 5: AST structure for the modified operator from Figure 3

for-loops form the main path nodes from start to end. Note that some paths have been left out for improved readability (green). Nested loops are visualized as sub-paths down to the nested computation. We use Algorithm 1 to traverse the AST and build dataflow graphs that follow the AST's topology and use the information found at its nodes. An initial Petri-Net graph that only has a start and end node connected on the backward path is created first (line 3). Then, sub-graphs for each node on the main path of the AST are sequentially inserted at the start node and connected to the end node when reaching the AST's end (line 10)). The sub-graphs are created similarly along the main node's sub-nodes, wherein the loop boundaries are used to determine the place sizes and token counts. Creating the dataflow graph from the AST enables adopting its topology, while paths marked by "parallel" or "pipeline" are created like in Figure 4. Visiting all loop nodes in the proposed structure allows tracking of their attributes and order. These can be provided to the hardware operations at the end of each AST sub-path. We find all information to build the dataflow graph nodes upon, like the type of operation, loop dependencies, and quantization, at the corresponding TVM operator. Together with the loop information, we can create hardware nodes like in Figure 3 and map these to valid VPs of our designated accelerator architecture model.
Automating the simulation graph's complete build process in the described structure allows directly generating simulation workloads from a CNN operator. Hardware-related modifications to the operator can be applied at the SW level and will be directly interpreted inside the transformation process.

VII. Evaluation

In order to show SimPyler's capabilities of flexibly generating latency simulation models and prove its application in DSE, we evaluate our methodology as follows. We deploy

Algorithm 1 Generate Dataflow Graph

Input: AST A
Output: DF Graph S
1: $start = A$.HEAD
2: $node = start$.next
3: $S =$ create_start_end_graph()
4: **while** $node$ is not A.TAIL **do**
5: sub_graph = create_node_graph($node$.attrs)
6: **for** sub_node in $node$ **do**
7: sub_node_g = create_node_graph(sub_node.attrs)
8: sub_graph.connect(sub_node_g)
9: **end for**
10: S.connect(sub_graph)
11: $node = node$.next
12: **end while**

two popular CNNs, VGG16 [14] and ResNet18 [8], on three different accelerator architectures realized on a Xilinx Zynq UltraScale+ XCZU9EG FPGA for our experimental setup. These architectures have been selected for the following reasons: 1) They are structurally similar to Eyeriss [3] and are comparable to state-of-the-art edge applications. 2) They are formally part of the same architectural design space and would provide possible candidates evaluated by SimPyler in DSE. 3) They differ in multiple architectural dimensions and come with individual dataflows, forming a solid baseline to show SimPyler's flexibility in representing various accelerators. We also show how the same VP-based simulation model can represent all three architectures while different dataflow graphs provide the intended functionality.

A. Baseline Accelerators

Figure 6 visualizes the coarse architecture of the proposed accelerators. We will refer to these as No-local-reuse (NLR), dual-core, and No-Output-Buffer (NOB), from a) to c). The names are derived from their functionality. NLR uses the on-chip buffers for data reuse, as in the no-local-reuse dataflow. Dual-core divides the computation of a single 32x32 core onto two main cores with dedicated buffers. Inside each core, the computation is again split between two 16x16 PE-Arrays. In NOB the output buffer can only store exactly one output pixel and so denies the natural overlap between computation in the PE-Array and storing of outputs into the DRAM by the DMA. For the NLR and the dual-core, all buses have a bus width of 128-bit. Input and output buffers are large enough to store all data for calculating two complete output pixel lines, while the filter buffer can store all filters of a CNN layer. In the case of NOB, the filter bus only has a bus width of 16-bit and the aforementioned output buffer size. Figure 6 d) depicts how all three can be represented in a single VP platform by SimPyler. All components are configurable to represent the hardware parameterization of the three implementations from before. Such architecture could be used in DSE, representing various architectural design points. Herein all components would be instantiated, but the dataflow graph will dictate which ones are

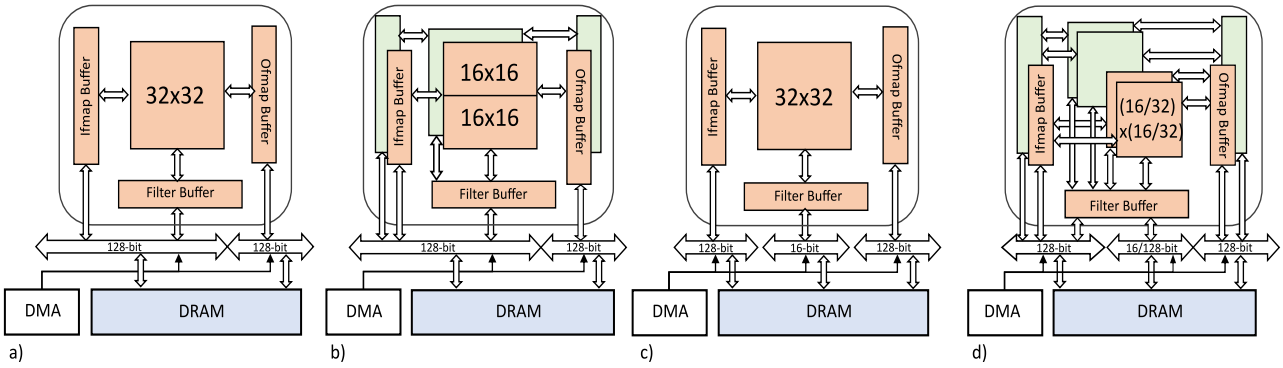

Fig. 6: a) No-local-reuse (NLR) accelerator: input/output buffer can store data to calculate two rows of output channels b) dual-core accelerator: dividing the computational load of a) between two main cores and two sub-cores, each buffer stores half of the data as in a), c) no-output-buffer (NOB): separate busses for all data types; the output buffer can only store one output channel, d) representing a) - c) in a unified VP-based SimPyler platform; "/" separated values refer to configurable parameters

really used for the simulation.

As we aim to prove SimPyler's capability of representing several dataflows and varying architectures, all accelerators in Figure 6 a) - c) are configured differently in how they execute certain CNN layers. Each accelerator provides different data transfer and computation schemes due to its architectural attributes, leading to individual temporal and spatial mappings of CNN operators. For evaluation, we model each accelerator using VPs and define the individual dataflow by creating the corresponding loop nest and hardware operations inside TVM schedules, as explained in Sections IV and V. SimPyler then automatically converts these inputs into executable simulation environments. By running the simulations, we gain latency values for the underlying architecture and dataflow. These are compared to the latencies measured onto the FPGA to determine SimPyler's estimation accuracy. Predictions are performed layerwise and only once for each layer due to the deterministic character of the simulation environments built by SimPyler. In the following section, we will validate our estimations' accuracy by simulating the baseline accelerators under the mentioned functionality.

B. Results

We simulate selected layers from VGG16 and ResNet18 on all three architectures. Further layers do not provide insights beyond the scope of the chosen ones and have been left out for simplicity. As the results were measured layerwise on the FPGA, we evaluated each layer individually.

Figure 7 shows the results we estimate for ResNet18. Overall, we achieve the latency within an error ranging from 0.9% to 5.8% and an average error of 3.69%. As the diagram shows, we predict the accuracy for most layers quite accurate. The errors herein correlate mostly to our DMA modeling. We express data transfer in NLR and dual-core by a linear DMA burst model because the accelerators process data row-wise and so have long data transfers. It is also beneficial that there are only a few conflicts between data transfer operations of the different data types, as filters are loaded independently from inputs and

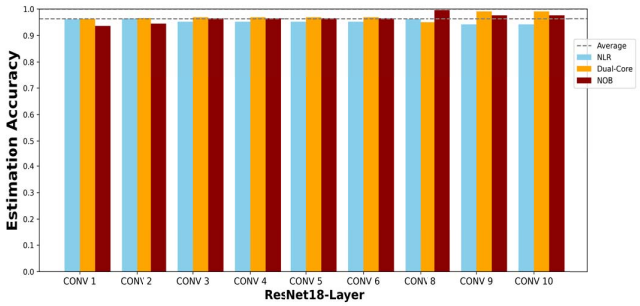

Fig. 7: Latency estimations for running ResNet18 on the three accelerator architectures NLR, dual-core, and NOB

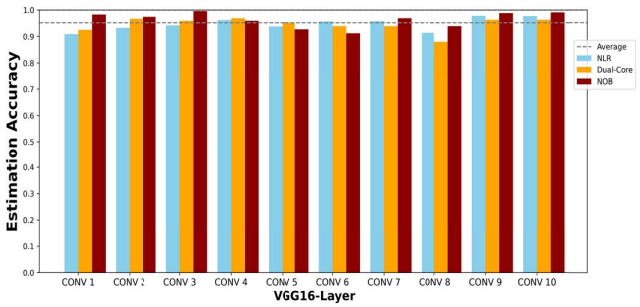

Fig. 8: Latency estimations for running VGG16 on the three accelerator architectures NLR, dual-core, and NOB

outputs, which themselves do not overlap much. This can be hard to predict without cycle-accurate modeling, which would invalidate our simulation's speed.

We see exactly that scenario in the case of the NOB. Memory conflicts can no longer be denied as filters and inputs are loaded in parallel. The comparably small bus width of 16-bit for the filter bus also prolongs filter transfers, resulting in even more conflicts with the simultaneous input data loads. This effect is visible in all layers but becomes more vital for layers 8 to 10 as the filter sizes increase in these layers, producing more memory conflicts. We still see high accuracy in our

estimations because we allow our DMA model to consider a fixed *conflict factor*, which is approximated from the amount of overlap between concurrent transfers. This factor will produce an additional penalty on the transfer latency in case of conflicts. Nevertheless, we still see some small errors as the DMA model only approximates this penalty and has some natural inaccuracy for conflict-less transfers. For the first five layers, this effect is comparably small. Herein, the latency is dominated by the storing of output pixels. This is because the accelerator performs the computation and transfer of outputs pixel-wise and in parallel. In Layers 1-5, it takes longer to store a pixel than to compute it, making the output transfer the dominant factor for the accelerator latency, while the computation dominates for Layers 6-10. As the results show, our modeling technique can adapt to these layers' bus requirements and precisely express the memory boundaries. For NLR and dual-core, all layers are mainly dominated by the computation inside the cores, and so the overhead of data transfers is only produced in the pre-load stage of inputs and filters wherein data is supplied to start the first computations inside the cores. For most of these, our DMA burst model achieves good estimations.

The previously mentioned effects also apply to VGG16 and are solved similarly. The results are depicted in Figure 7. Herein we observe an average error of 4.8%, within a range from 0.9% to 12%. We use our methodology as before, observing that we precisely model the individual layer's boundaries to computation or memory effects. Only layer eight on the dual-core suffers from some inaccuracies, with an error of 12%. This error is also related to the data transfer, primarily due to our DMA model. The first layer's latency in the NOB relies again on the short computation time for convolution outputs and the comparably long storage time, which we predict correctly with errors of less than 7%. We overcome later layers inaccuracies caused by inconsistent memory conflicts with our DMA adjustments, seeing even an increase in estimation accuracy. NLR and dual-core results are dominated by computation, which our methodology expresses as well, proved by the low prediction errors.

In summary, we show that SimPyler can predict computation and memory boundaries on three different architectures with different dataflow. Even dynamic memory effects, as bus conflicts and dominating data transfer times, are estimated accurately. SimPyler's simulation character enables expressing these scenarios precisely, while the embedded analytical models provide enough precision to forecast the system's latency with high confidence. In our experiments, we observe that most layer's simulations execute in < 30s, while we see only a few outliers, which is slower than analytical approaches but comparably fast for simulation. This timing includes the simulation setup of initilaizing the VP platform, creating the dataflow graph, and mapping to the VPs.

In addition, SimPyler can perform several simulations in parallel by leveraging multiple cores of the system it is executed on. This can speed up the evaluation significantly in a potential DSE application.

VIII. Conclusion

In this work, we proposed SimPyler, a simulation-based framework for estimating the latency of CNNs on ML accelerators. By executing dataflow graphs onto architectures built from virtual prototypes, SimPyler addresses current challenges for latency predictions from the HW and SW perspectives. Leveraging the machine learning compiler framework, TVM allows for automating the processes of creating the graph and architecture mapping, enabling SimPylers application in DSE. We can estimate the error for running common CNNs onto multiple accelerators with an average error of 4.12%.

Our methodology allows for integrating SimPyler into TVM's auto-tuning tools AutoTVM/MetaSchedule to perform design space exploration based on SimPylers estimations.

Acknowledgment

This work has been funded by the Federal Ministry of Education and Research (BMBF) project MANNHEIM-FlexKI.

References

[1] Tianqi Chen et al. Tvm: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[2] Tianshi Chen et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[3] Yu-Hsin Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

[4] Yu-Hsin Chen et al. 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):292–308, 2019.

[5] Ronan Collobert et al. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[6] Shail Dave et al. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–27, 2019.

[7] Zidong Du et al. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

[8] Kaiming He et al. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] Sunwoo Kim et al. Optimization of multi-core accelerator performance based on accurate performance estimation. *IEEE Access*, 10:19629–19642, 2022.

[10] Michael J Klaiber et al. An end-to-end hw/sw co-design methodology to design efficient deep neural network systems using virtual models. In *Proceedings of the INTelligent Embedded Systems Architectures and Applications Workshop 2019*, pages 18–22, 2019.

[11] Alex Krizhevsky et al. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[12] Linyan Mei et al. Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. *IEEE Transactions on Computers*, 70(8):1160–1174, 2021.

[13] Angshuman Parashar et al. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[15] Sam Xi et al. Smaug: End-to-end full-stack simulation infrastructure for deep learning workloads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.

[16] Alexandre V Yakovlev and Albert M Koelmans. Petri nets and digital hardware design. *Lectures on Petri Nets II: Applications: Advances in Petri Nets 3*, pages 154–236, 1998.