Improving mapping of convolutional neural networks on FPGAs through tailored macro sizes

Brindusa Mihaela Damian-Kosterhon, Andreas Koch
Technical University of Darmstadt
Darmstadt, Germany,
damian@esa.tu-darmstadt.de, koch@esa.tu-darmstadt.de

Felix Kosterhon
SECUINFRA GmbH
Berlin, Germany

Lucian Petrica
AMD
Dublin, Ireland

***Abstract*—Implementing convolutional neural networks (NNs) on FPGAs has become more popular as top-down flows allow designers to describe NNs in high-level languages and easily generate the corresponding FPGA bitstream. However, bitstream generation is time-consuming, leading to long development times in case of NN adjustments. This can be addressed by flows that split the top design into blocks, cache their implementation, and stitch them together to obtain a full bitstream. This way, modifying only one block does not require the re-implementation of the others. Yet, such algorithms don't perform well for large designs using most of the FPGA resources, which is often the case for NNs. We show that, in such a scenario, the block's size is essential, and for a sample quantized convolutional NN with the typical convolutional, fully connected, and max pool layers, it leads to 15% more placed blocks. Therefore, we develop an estimator for the block's area constraints, which we successfully use on the evaluated design to place it 1.37 times faster.**

***Index Terms*—FPGA, Floorplanning, PBlock, RapidWright, Neural Networks**

I. Introduction

In scenarios where using a discrete GPU or other ML accelerator is not practical, e.g., in lower-cost / space constrained embedded scenarios that already rely on a reconfigurable SoC (rSoC) with an integrated FPGA, such as the Xilinx Zynq or UltraScale+ MPSoC [3], as main control unit, NNs can be realized efficiently on these rSoCs. In general, the energy efficiency to execute the NNs in the reconfigurable logic on these devices is far better than on their standard processing cores, especially when custom numeric types (INT4/2/1, FP8/4) can be employed in the NN that are not supported on the currently used rSoC CPU cores, such as ARM Cortex-A9. Furthermore, FPGAs are often used as a prototyping vehicle before committing an NN into an ASIC for large-volume deployment in systems that do not need the additional capabilities of a GPU or programmable ML accelerator.

A major challenge when using FPGAs for NN inference is that NNs often require more resources than those available on device [24]. Strategies to overcome this include using lower precision weights [25] or storing them externally [15]. While the first one often reduces the accuracy, the second one increases the inference run-time. Fortunately, the growing interest in FPGAs [2] led to larger devices [3], which in turn facilitates the mapping of larger NNs. Since implementing a design on an FPGA is an NP-complete problem [23], larger designs lead to significantly longer compilation times. This conflicts with the fact that FPGAs are often used in early design phases for *quick* testing of different architectures. However, if changing some parts of an NN leads to high compilation run-times, FPGAs become unattractive.

Existing electronic design automation (EDA) tools provide different mechanisms to reduce the tool run-time for large designs (e.g., incremental flows [5], [13]). However, their speedup is usually achieved only for small design changes. For example, an industrial EDA tool optimized for mapping to AMD FPGAs can achieve a 2x speed-up if at least 95% of the design is reused [5]. Unfortunately, the required NN design updates are often larger, as architecture changes in one layer also influence the wiring and processing of adjacent layers.

Another possible approach is to utilize dynamic partial reconfiguration such that only updated logic must be rebuilt. This is a powerful feature, as the user can load the modified bitstream of a single module, while the rest of the design is still running. Yet, the design and FPGA area must be partitioned offline, and the updated module might have a much higher or lower resource usage than the assigned FPGA area. In the first case, the reconfiguration is unfeasible. In the latter one, the module uses fewer resources than assigned, wasting area.

Another alternative for speeding up the implementation of large designs is given by algorithms that make use of *pre-implemented* blocks, such as RapidWright (RW) [18]. RW assumes that an FPGA design is constructed from multiple *blocks/modules* connected together in a diagram. The tool identifies unique block configurations and implements each of them independently, resulting in pre-implemented blocks, which are relocatable placed and routed netlists. Afterwards, it replicates and places on device all the pre-implemented blocks as necessary to reproduce the original diagram, and connects them to obtain a full bitstream. With RW, if only a single module needs to be modified, re-implementing the others is not required, thus speeding up the compilation.

Algorithms based on pre-implemented blocks have drawbacks as well. While separately implementing each module, RW constrains its placement inside of a placement area block (*PBlock* in AMD terminology). The PBlock size highly impacts the quality of the final placement: If the PBlock constraint is too loose, the implemented module could waste area, and if the constraints are too tight, the module's placement and routing could fail. The first contribution of this paper

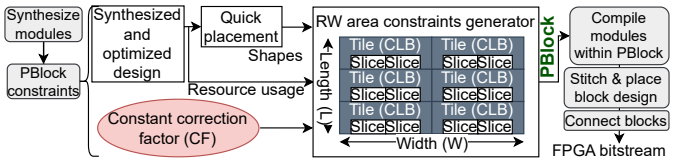

Fig. 1. RapidWright (RW) algorithm for the PBlock area constraints.

is implementing the *cnvW1A1* NN [1] for the RW-flow to show that PBlock constraints impact the allowed NN size for a feasible FPGA placement. It is a VGG-inspired quantized network with the typical convolutional, fully connected, and max pool layers, such that the concepts shown here are transferable to other such convolutional NNs.

While the first contribution of our work highlights the importance of the PBlock size, the second contribution focuses on developing a PBlock estimator. In order to evaluate the required number of FPGA resources for a given module, RW synthesizes and optimizes the design and runs a fast placement. Based on the information gained from these runs, it estimates the size of a valid PBlock, as shown in Figure 1. A valid PBlock constraint has sufficient resources, such that the module is fully placed and routed within its boundaries. Otherwise, the flow will stop. To avoid such a failing run, RW applies a correction factor (CF) to the resource number. However, since this factor is a constant, it is not tailored to the design being implemented, and results are often sub-optimal. The second contribution of this paper is an estimator for this CF, which leads to faster implementation run times during the stitching process, as well as better quality of results, most notably allowing the mapping of larger designs. We evaluate different machine learning algorithms, among which the best result has a relative error below 5%.

II. Related Work

There are multiple approaches for speeding up FPGA compilation. One solution is to tweak the compilation algorithms. An alternative is to reduce the size of the circuit to be compiled through the re-use of pre-compiled modules [20]. Our work focuses on the latter one, as our application scenario represents the design space exploration of NNs, where the user re-compiles the design after modifying only parts of the neural network.

The community already offers some solutions using pre-compiled modules. PRFlow [9] is such an example and uses partial reconfiguration. The design consists of multiple partitions which can be independently recompiled at run-time. They use a packet-switched network for the communication between the partitions. On the one hand, such a communication network becomes a critical overhead as our NN-based application scenario already utilizes most of the FPGA resources. On the other hand, in the case of partial reconfiguration, the partitions are defined at compile time and can not be resized at run-time. Adjusting them to fit the updated module can only be done offline, which brings us back to the class of tools generating the full bitstream at compile-time, such as HMFlow [20]. It proposes a flow that separately compiles all the blocks of a design and stitches them together to form a full FPGA bitstream. For interacting with the FPGA vendor's format, HMFlow uses RapidSmith [21], which was further continued as RapidWright [18] (RW). It has Java classes for communicating with the vendor's format but also to facilitate an HMFlow-like flow with pre-compiled modules. DynaRapid [12] is an example project using RW to speed up bitstream generation for the elastic circuits created by the Dynamatic HLS tool [14]. While RW uses the AMD format, BPR [7] is a tool using pre-compiled circuits, but for the Intel format. We use RW in this work, as it is currently well supported, and the main concepts can be replicated also to other FPGA vendors.

Although RW can speed up compilation [16], it poses some challenges when placing large designs, as we will show in Section IV. This is partly due to the area constraints applied when individually implementing the modules. In RW, the pre-compiled modules are placed and routed within a rectangular bounding box, called in AMD terminology a *PBlock*. It consists of multiple slices, which are made of different resources, such as LUTs, MUXes, and FFs. As we deal with large designs, estimating the PBlock size for efficient resource utilization is the main topic of this paper.

There are already numerous approaches for estimating FPGA resource usage. Among the ones that address estimations based on a high-level specification of the design, some do not take into account synthesis optimization [11], leading to low accuracy. Other focus only on certain components (e.g., shifters, adders, multipliers) [22], or estimate the resource usage in LUTs and register numbers [6]. In contrast, we focus on estimating the geometric PBlock area constraints, while the LUTs and FF numbers are already known from the synthesis results. An estimator for the FPGA slice number is given in [8], but it focuses only on certain application examples (e.g., floating and fixed point IP cores). A more accurate model is proposed in [28] based on a predicted LUT mapping, which unfortunately leads to long run times. In contrast, [26] targets a fast slice number estimation, but their results are based solely on LUTs and FFs. In our work, we show that also other factors can impact the PBlock size, like FF resets, fanout, or carry chains. Last but not least, we target FPGA slice number estimation for PBlocks, which by definition have rectangular shape constraints, and this is not taken into consideration in any of the work above.

There are some works highlighting the importance of the selected PBlock size as part of a RW-like flow. They observe its impact on frequency [19], [7], and block relocation during the stitching process [17], but they don't propose and analyze a PBlock estimator, or they use a simple model based solely on the LUTs and FFs number [27]. During our analysis, we observed that this is not enough for an optimal PBlock estimator; hence, we searched for an improved model.

III. Application Scenario

The community offers end-to-end flows for mapping NNs to FPGAs. One such example is the FINN [10] framework. The user defines an NN in Python, which undergoes multiple transformations and, in the end, is compiled to a bitstream that can be run on an FPGA. Modifying the NN in the search for the best configuration is fast, thanks to the high-level languages. However, during this process of design space exploration (DSE), FPGA compilation becomes a bottleneck. Through traditional flows, updating one part of the NN would require re-compiling the entire design. It was shown in [16] that this could be accelerated through rapid prototyping tools like RW, which make use of pre-compiled modules. As this was already proven in prior work, we will not focus here on the run-time but rather on another challenge we confronted while using RW. During our experiments, we encountered the issue that convolutional NNs usually take up many resources and RW has difficulties placing designs that use most of the FPGA's area. One solution is using a larger FPGA. However, during DSE, switching between FPGAs to match RW requirements is sub-optimal. To illustrate the challenges and proposed solution, we showcase a NN using most of the FPGA resources, and we design it for the RW flow.

Because FINN provides an end-to-end flow facilitating fast DSE, we chose one of its supported NNs, namely the *cnvW1A1* network [1], aiming to achieve a fast NN architecture exploration by leveraging the speed ups brought by RW. However, the circuit generated through FINN was a monolithic block, while RW expects as input a block design with multiple interconnected blocks. The granularity, i.e., coarse or fine, of these modules impacts the tool's run-time [19]. One approach would be to divide the NN layer-wise, thus having a block per layer. However, each layer would be instantiated only once in the design. Thus, the placed and routed netlist of one block would not be reused for other instances, which is the main selling point of RW. Yet, NNs usually have a regular structure. To use this to our advantage, we select the design's granularity such that we have separate blocks for implementing the matrix-vector activation unit (MVAU), the sliding window, the activation, as well as the max pool units. The NN has nine convolutional and fully connected layers, and two max pool layers (Figure 2). Its structure is typical for convolutional NNs such that the concepts presented here are transferable. The partitioned design has a total of 175 blocks. Thanks to the NN's regularity, the number of unique instances is lower, namely just 74. The highest reuse is for the MVAU units. Layers one and two have the same MVAU configurations (48 identical instances), as do layers three and four (20 identical instances). This high reuse of the same module highlights that this is a valid application case for a flow with pre-implemented modules such as RW. In general, such high reuse is typical for convolutional NNs, which commonly have multiple instances of the same processing element within a layer. Unfortunately, RW is not able to fully place the sample design, although the *cnvW1A1* is one of the smaller NNs using only 1 bit for weights and activation [1]. The causes are analyzed in the following section.

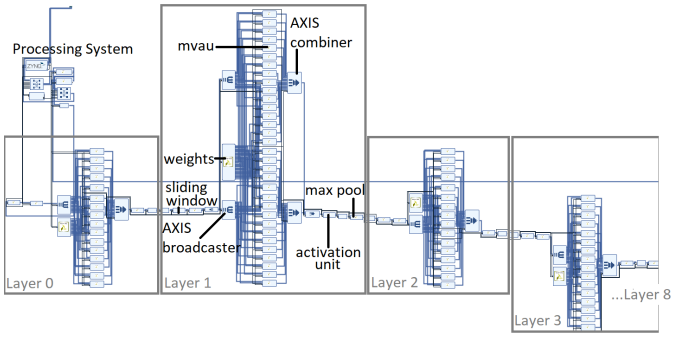

Fig. 2. Block design of the implemented *cnvW1A1* NN.

IV. The Importance of a PBlock Estimator

RW was not able to fully place the *cnvW1A1* design on a xc7z020 device. Among the causes, we identified that the PBlock had a huge impact, which we demonstrate here.

For our analysis, we first evaluated the *cnvW1A1* NN on a xc7z020 device by compiling it with the EDA tool for AMD FPGAs, version 2024.2. The tool was able to fully place and route the entire design. The corresponding implementation is shown in Figure 5a) and uses 99.98% of the device slices.

Subsequently, we implemented the same block design also with RW. In order to generate a bitstream, the original RW algorithm first identifies the unique blocks in the block design. It then synthesizes and optimizes each of them separately. Afterward, it creates rectangular area constraints (*PBlocks*) for each block, which will later be used as a bounding box for the place and route step. These area constraints are estimated as shown in Figure 1, based on resource usage, and geometric shapes of a quick placement. The estimated slices are then multiplied by a constant correction factor (CF) of 1.5 to obtain the final PBlock shape. After the modules are individually placed and routed within the PBlock, RW stitches them all on the FPGA to form a complete design bitstream.

The flow did not achieve a full placement, and while analyzing the results, we noted that it was mainly due to the floorplanning of the individual modules, which motivated this paper. We observed many unused slices between the blocks. This can be explained by visualizing the separate placements of the modules. Figure 3 illustrates the placement of the largest block in the design (`weights_l4`), with 1529 slices, and of a much smaller one (`mvau_l8`), with 31 slices. The constant PBlock CF of 1.5 leads to irregular shapes for both modules. When the stitcher places all separately implemented blocks on the device, it checks for overlapping logic. Naturally, the more irregularities there are in the separate placements, the harder it is to bring them closer. This leads to "dead spots" in the stitched placement, although there still are unplaced blocks. In contrast, when implementing these blocks within the smallest feasible PBlock, their placement becomes more

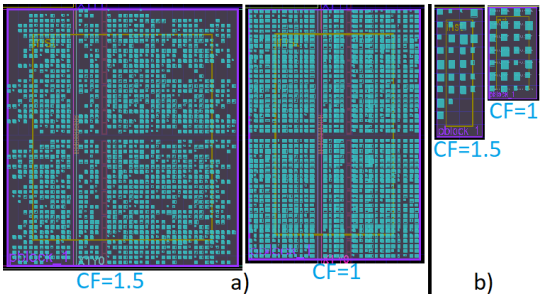

Fig. 3. Implemented blocks with a constant PBlock correction factor (CF) of 1.5 versus 1 for the: a) weights_l4 and b) mvau_l8 modules.

TABLE I
SYNTHESIS RESULTS OF THE *cnvW1A1*

	RW PBlock Slices		RW PBlock Longest Path (ns)		AMD EDA Slices
CF*	1.5	1	1.5	1	-
mvau_l8	31	28	4.829	5.769	30,34,32,29**
weights_l4	1529	1371	10.767	13.478	1430

*CF = Correction Factor **mvau_l8 has four instances. RW reuses a single implementation for all four, AMD EDA implements each of them

rectangular. This also helps the simulated annealing algorithm of the stitcher to reduce the illegal moves caused by overlaps.

Apart from the placement regularity, it can be observed that when implementing the module with the smallest feasible PBlock constraints, fewer slices are used (Table I). A more compact PBlock can be placed more easily to avoid clock distribution columns, which worsen the circuit's timing [19]. Note that looser PBlock constraints often lead to more used slices than the AMD Tool. This is probably because the latter uses 99.98% of the available slices, forcing the algorithm to optimize area. While the utilization improves with harsher area constraints, the timing is affected. Still, the main issue is the incomplete placement of the entire design, such that the circuit's timing is a secondary criterion.

We further conducted an analysis in which we determined with a 0.02 resolution the minimal CF of the NN's modules, for which the place and route is still feasible. Figure 4 shows the resulting distribution for *cnvW1A1*. The values below 0.7 correspond either to very small modules or to modules whose area constraints are driven by the block RAMs. In both cases, further reducing the CF would not change the actual PBlock size. The highest CF was 1.68. In a setup using a constant CF, the user must set it to 1.68, such that all modules have a feasible implementation and the tool flow does not stop. In this case, the example NN has 68 remaining unplaced modules after the stitching step (Figure 5b). In contrast, using the optimal CF (tightest feasible PBlock) for all modules leads to 52 unplaced modules (Figure 5c). The highlighted instances correspond to the module in Figure 3b, and they are better integrated into the final placement when using the optimal CF. The remaining unused FPGA columns exist because PBlocks can be relocated only on columns having the same resource type, which again favors more compact PBlocks.

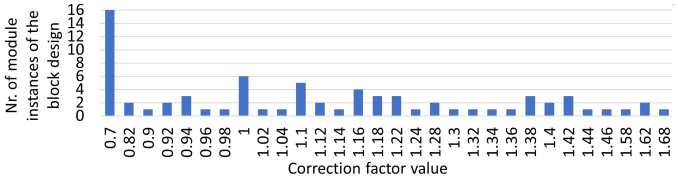

Fig. 4. Distribution of the optimal CF for the blocks of the *cnvW1A1* design. The vertical axis shows the number of blocks which have the CF on the horizontal axis.

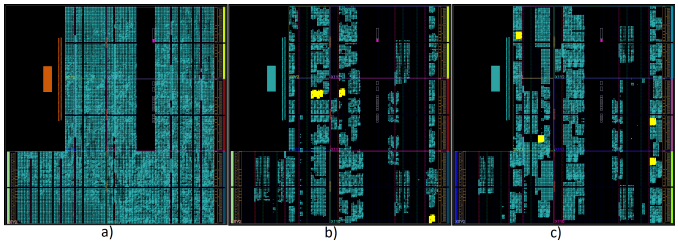

Fig. 5. Placed and routed *cnvW1A1* using a) AMD EDA tool b) RW with a constant PBlock CF of 1.68, and c) RW with a minimal feasible CF.

V. FACTORS IMPACTING PBLOCK SIZE

To build an accurate PBlock estimator, we analyzed the factors impacting the PBlock size for the Zynq 7 FPGAs. These will be used as features in our estimator.

A. CLB type

AMD FPGAs have a regular structure, and one of their basic building blocks is the configurable logic block (CLB). It consists of slices, which can be of either L or M-type. They differ because the latter also supports shift registers and distributed RAMs. M-type CLBs have an M-type and L-type slice. A design requiring many M-type slices will automatically infer many M-type CLBs. As these also contain an L-type slice, the module will implicitly be assigned additional resources, which must be considered when sizing the PBlock.

B. Control set conflicts

Any SRL, LUTRAM, or register is controlled by a set of signals such as reset, clock, or enable. The grouping of such signals is called a control set [4]. Two registers belonging to different control sets can not be placed within the same CLB. Therefore, when selecting the CF, we must consider that some CLBs might be under-utilized due to control set conflicts.

C. Carry Chains

A carry chain requires a specific shape for the area constraints, which is given in the shape report generated by RW after the quick placement (Figure 1). Without it, the algorithm could generate the wrong PBlock width and height, and lead to an infeasible placement despite having sufficient resources.

D. High fanin/fanout signals

High fanin/fanout signals imply more routing effort, which typically requires larger PBlocks for more routing channels.

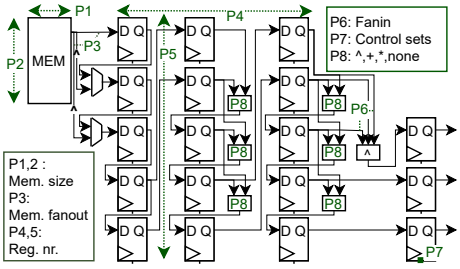

Fig. 6. Template for generating the RTL data set for the PBlock estimator.

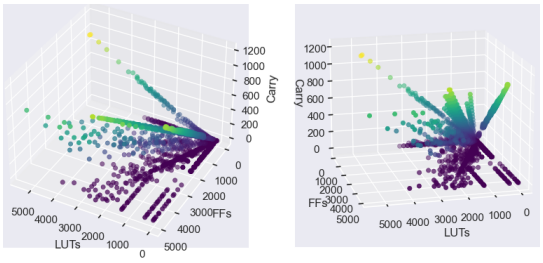

Fig. 7. Design space coverage based on the RTL generators. The three-dimensional diagram shows from two perspectives the LUTs (green), FFs (purple) and Carry (yellow) usage.

E. High density designs

A slice of the 7-series device contains four LUTs, one carry chain segment, and eight FFs. If a module requires the same number of slices for the FFs, LUTs, and carry elements, the probability that all of them will be packed in the same slices is lower due to congestion. Thus, the PBlock will probably need a higher CF.

VI. Implementation

This section covers two aspects: data set generation and possible implementations of an improved PBlock estimator. Further estimator characteristics are discussed in Section VII.

A. Data Set Generation

For an accurate estimator, the training set is crucial and must cover all the particularities highlighted in Section V. Instead of varying the implementation of the *cnvW1A1* modules, we approached a more generic solution and implemented multiple register-transfer level (RTL) generators.

One of the generators covers the corner case of a design containing *mostly* FFs. It consists of shift registers with a parametrizable number of control sets and fanin. A tool attribute was used to avoid their implementation within LUTs. In contrast, the second generator focuses on modules with *no* registers at all, having mainly LUTRAMs. The memory's width and depth are parametrizable. The third generator focuses on carry chains. It consists of a basic sum of squares with parametrizable data widths. The fourth generator aims to use FFs, LUTs, carry, and shift registers and is implemented as multiple linear-feedback shift registers (LFSRs). The rest of the RTL generators contain all the resources mentioned above and are parametrizable. One example is illustrated in Figure 6. Although it does not have a meaningful application, its purpose is to cover as much of the design space as possible.

In total, the generated data set contains around $2,000$ modules, and the resource usage distribution is shown in Figure 7. The largest modules have around 5000 LUTs (11% of the device). We did not generate larger modules, as RW yields the best speed-ups for partitioned designs with high regularity. Thus, larger blocks would not fit this scenario well.

B. Selecting an estimator type for the CF of the PBlock

This paper considers four approaches for the estimator of the CF used in the PBlock generator (Figure 1). One employs linear regression, the second relies on a NN, while the last two are tree-based.

For linear regression, we used the following parameters which were introduced in Section V. The first two are given by the maximum fanout and the number of control sets. The third one is denoted here as *PBlock density* and it encompasses the idea that modules having slices filled with all three types of resources (LUTs, FFs and carry) commonly require higher CFs compared to designs with one resource type (Section V-E). The fourth parameter is the ratio between the required M-type slices and the estimated total amount of slices. Another input parameter focuses on the ratio between the carry cells and the estimated total number of slices. The last input parameters are gained from the shape report described in Section V-C.

The second implementation of the estimator is NN-based. A shallow feed-forward network with only one fully connected hidden layer is used due to the small number of inputs. We varied the number of hidden neurons finding that 25 neurons provide robust results for our training set. We are using a fully-connected NN instead of a convolutional NN as the inputs represent distinct features. Dropout was considered but not used in the final design, as it did not improve the training and test performance. The common activation function ReLu (rectified linear unit) is used to add non-linearity to the model. Finally, the optimization algorithm ADAM [14] adapts the weights of the network to minimize the mean squared error (MSE) between the predicted and the actual minimal CF.

Additionally, we implemented a random forest (RF) based solution. It combines the predictions of 1000 decision-trees (each with a depth of 20) to estimate the minimal CF. The mean squared error (MSE) was used to train the RF. Compared to the NN approach, the expressiveness of the RF seems rather limited. However, while we had to deal with a black box in the NN, the RF approach provides interesting insights into the decision-making and into the relevance of the different features. While the implementation itself might be of interest we consider the evaluation of the feature importance (see Section VII) as most relevant for further research. By analyzing the importance of the different input parameters in the decision making, we note an advantage of relative parameters.

We further decreased the estimator complexity by using a single decision tree (DT) with a depth of 20 and analyzed the importance of the carefully selected features (see Section VII).

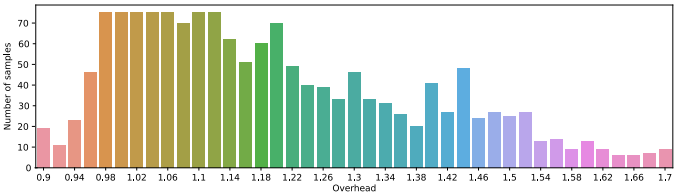

Fig. 8. Distribution of the input data over the correction factor.

TABLE II
RELATIVE ERROR OF THE PROPOSED ESTIMATORS

Features	Classical	Classical*	Additional	All
Decision Tree Error	7.4%	7.4%	5.4%	5.2%
Random Forest Error	6.2%	5.9%	4.8%	4.9%
Neural Network Error	-	-	-	5.1%

C. Resolution

When searching for the optimal CF, the magnitude of the search step is essential. It was observed that designs with less than approximately 100 LUTs would not require a search step lower than 0.1. Because of the constant PBlocks aspect ratio (W/L in Figure 1), a circuit's modification smaller than 10% would not make a difference for such small designs. However, it was observed for designs having around 2,500 LUTs that the CF should not be higher than 0.03. As 85% of our data set has less than this resource usage, we selected a search step of 0.02. If the feature set is selected carefully, the results will be comparable for a lower magnitude of the increment step.

VII. EVALUATION

Before testing the estimators, the quality of the input data must be evaluated. After generating the RTL data set illustrated in Figure 7, the minimum CF (Figure 1) is determined by starting from a factor of 0.9 and increasing it by 0.02 till a feasible placement within the PBlock is found. Unfortunately, the obtained CF has an uneven distribution. This is also due to the fact that during parameter sweep, some modules have generated more instances than others. In order to avoid focusing the training process only on a specific area in the design space, we filtered the input data to obtain a more even distribution. Still, our dataset remains biased, which is important for the following evaluation. An upper limit of 75 samples was set for each CF after shuffling the data. The distribution of the input data is shown in Figure 8. By setting an upper limit, the data set was decreased from 2,000 to 1,500 samples. For all four PBlock estimator approaches, 80% of the data was used for training and the remaining part for testing.

The linear regression function described in Section VI-B taking nine inputs returned a mean relative error of 9.4%. It is significantly higher compared to the other proposed algorithms, which are shown in Table II.

Table II illustrates the relative errors (minimal versus estimated CF) of the NN, RF, and DT-based solutions, while Figure 9 shows the impact of each chosen feature set. The "Classical Features" ("Classical" in Table II) contain the number of LUTs, CLBMs, FFs, control sets, and carry elements, as well as the maximal fanout. The extended "Classical Features with Placement Features" ("Classical*" in Table II) additionally uses the estimated shape area, obtained from the quick placement in RW (Section V-C). To develop size-invariant features, the "Additional Features" ("Additional" in Table II) use relative amounts, and their expression is given under the green bars in Figure 9. In the end, we also use all the features together as "All Features" ("All" in Table II). While the errors of the RF and DT algorithms are shown for each feature set, we fed all our features to the NN to get the best possible performance. We make the following observations:

- The RF approach is slightly better than a single DT, which can be reasoned by the fact that the RF combines the estimated output of N (set to 1,000) trees instead of using just one.
- Our proposed "Additional Features" outperform the (extended) classical features in both approaches and are invariant to the circuit size.
- The placement features do not seem to improve the performance significantly compared to other features.
- For RF, the precision of our "Additional Features" is not improved by using all of the features (adding absolute numbers). For the DT approach, the difference is small.
- Comparing the relative mean error, the NN approach provides a similar accuracy as the two tree-based solutions. Thus, increasing the expressiveness of our estimator does not always lead to better results.

To get a better understanding of our implemented estimators, we show in Figure 10 the predictions along with the true correction values (green line). The estimates for the different CFs confirm our previous observations. Our proposed "Additional Features", as well as "All Features", clearly perform better than the classic features. This is observed in particular on high CF values. Considering our biased data set, it is harder for learn-based algorithms to predict high CFs. Still, our proposed relative features seem to overcome this challenge by reaching an error below 5% for the generated data set.

Even more interesting than the achieved mean error is the fact that the most successful feature-set, "Additional Features", contains only relative values (given in Figure 9). The red bars are the most dominant for the relative features, which shows that these are the preferred ones even when all features are available. For each feature-set, the sum of the importance values is 1. Note that the relative number of Carry elements (Carry/All) has a value of 0.5 for "Additional Features" and of 0.4 for all features. Thus, even when having all the available features, this single one makes up 40% (0.4) of the decision.

In the end, our evaluation proves the effectiveness of learning-based strategies by reaching an error below 5% for the generated dataset. Furthermore, hand-crafted relative features clearly outperformed the raw input data, motivating further research.

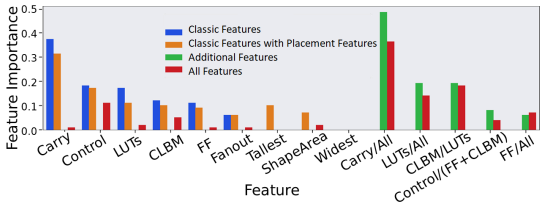

Fig. 9. Importance of different features when using a single DT. The sum of the importance of all features is 1. The importance of 1 for a single feature would mean that the output solely depends on this input.

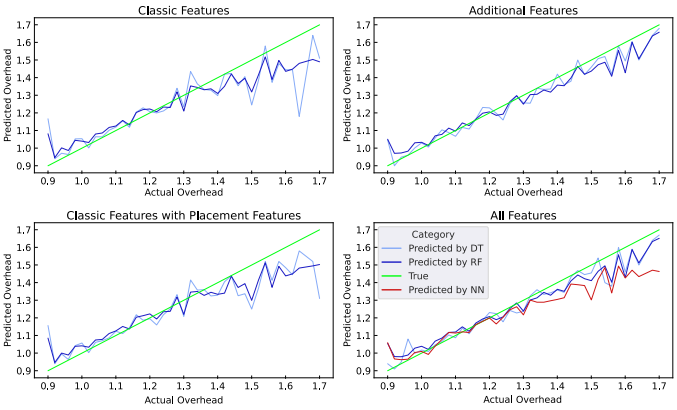

Fig. 10. Predicted versus actual (minimal) correction factor value.

VIII. Estimator Impact

The sparse placement of the *cnvW1A1* NN on the FPGA motivated the need for a PBlock estimator, and here we evaluate its impact on the NN.

First, we implemented the blocks of the *cnvW1A1* NN using the different estimator types. We removed the modules that had one or two tiles from the evaluation, as their PBlock is straightforward and they do not require an estimator. Figure 11 shows in orange the estimated versus the actual CFs for the 63 implemented modules when using linear regression (median absolute error of 11.03%). Figure 12 confirms the observation from Section VII that the "Additional Features" contribute more to the evaluation, thus we used them also for the NN-based estimator (median absolute error of 9.5%).

Secondly, we analyzed how such an estimator impacts the final placement of the design. Our goal is to obtain tight area constraints with minimal effort, having two main criteria:

- run-time: we used RW to speed up compilation time, therefore, this still remains an essential criterion

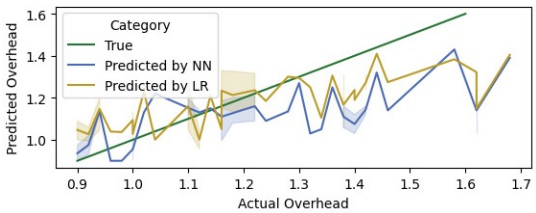

Fig. 11. Actual versus estimated CF when using linear regression.

Fig. 12. Importance of different features when the *cnvW1A1* is used as a test set for the RF approach. The sum of the importance of all features is 1. The importance of 1 for a single feature would mean that the output solely depends on this input.

- PBlock density: we aim for smaller PBlocks with a compact placement of the module, such that we can place more instances on the FPGA (motivated in Section III)

When using the NN estimator on the *cnvW1A1* design, 52.7% of the modules are implemented from the first run. For the underestimated CFs, we increment the correction factor by 0.1 and when a feasible correction factor is found, the last interval is searched with a resolution of 0.02. If we compare our estimator-based approach with a constant-CF approach, where the initial CF=0.9 (to aim for compact PBlocks), the version using a constant CF needs 1.8 times as many tool runs as our estimator-based approach for block compilation.

An estimator that mainly underestimates the CFs will lead to more tool runs, but provide more compact, area-efficient PBlocks. Therefore, by adding an overhead to the estimator, the user can adjust which of the two goals (run-time versus PBlock density) is more critical to the application scenario.

Of the 63 estimated CFs, 31.75% have an error below 4% when compared to the minimal CF, leading to compact PBlocks. We further show how this impacts the full placement of the design on an xc7z045 device. RW uses simulated annealing (SA) to place all the pre-implemented modules and reconstruct the block design. The RW SA cost function is defined based on the wire length between the blocks, which, naturally, should be small. As was elaborated in Section IV, tighter PBlocks make the placement of the individual modules more rectangular, which eases bringing the blocks closer to each other and reduces the number of illegal steps in SA. With our estimator, the SA algorithm converged 1.37 times faster. Moreover, the final SA cost was 40% lower when using a PBlock estimator, versus a constant CF of 1.68 (1.68 was motivated in Section IV). This can also be visualized in Figure 13, where the placement using an estimator has less wasted area between the blocks. This again highlights the importance of a PBlock estimator in a flow using pre-compiled blocks. Apart from the PBlock size, an important aspect is its position, which leads to the empty columns in Figure 13. However, their position is not studied here and is of interest for future work.

IX. Conclusion

Mapping NNs on FPGAs is challenging due to the high resource usage and compilation times. In this work, we partitioned the *cnvW1A1* NN to compile it with a rapid prototyping technique using pre-implemented modules. We

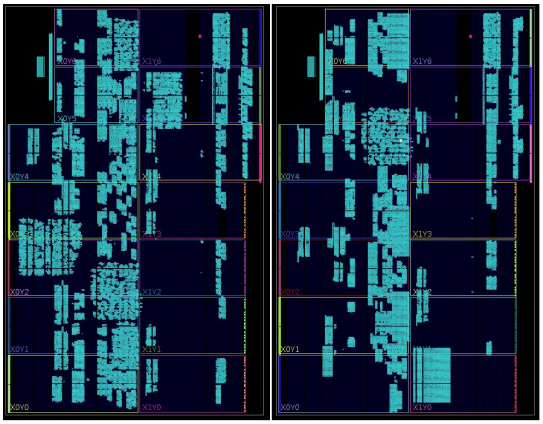

Fig. 13. Comparison of the fully placed *cnvW1A1* when using a constant CF of 1.68 (left) versus our proposed estimator (right).

showed that the size of the actually placed design depends on carefully selecting the area constraints (PBlock) for the individual design blocks. These are computed based on the resource usage, multiplied by a correction factor (CF) for which we implemented an estimator using four approaches: linear regression, NNs, decision trees, and random forest. Our data set contained designs with resource usages between 12 and $5,000$ LUTs and a CF between 0.9 and 1.7. Through our estimator, the global placement algorithm converged 1.37 times faster and with a lower cost. Our findings bring us closer to mapping large NNs to FPGAs using rapid prototyping techniques with pre-implemented modules.

X. Acknowledgment

The support of AMD for this research is greatly appreciated.

References

[1] cnvW1A1 as part of the BNN-PYNQ Project . https://github.com/Xilinx/BNN-PYNQ. accessed Dec. 2024.

[2] Field Programmable Gate Array (FPGA) Market Size, Share and Trends Analysis Report, 2023 - 2030. https://www.grandviewresearch.com/industry-analysis/fpga-market. accessed Dec. 2024.

[3] AMD. AMD FPGA Portfolio. https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc.html. accessed March. 2024.

[4] AMD. UltraFast Design Methodology Guide for the Vivado Design Suite. v2012.2, June 2007. pp 48.

[5] AMD. Vivado Design Suite User Guide: Implementation (UG904). https://docs.amd.com/r/2024.1-English/ug904-vivado-implementation/Incremental-Implementation. accessed Dec. 2024.

[6] C. Brandolese, W. Fornaciari, and F. Salice. An area estimation methodology for fpga based designs at systemc-level. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 129–132, 2004.

[7] James Coole and Greg Stitt. Bpr: fast fpga placement and routing using macroblocks. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, page 275–284, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Lanping Deng, Kanwaldeep Sobti, and Chaitali Chakrabarti. Accurate models for estimating area and power of fpga implementations. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 1417–1420, 2008.

[9] Xiao Yuanlong. et al. Reducing fpga compile time with separate compilation for fpga building blocks. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 153–161, 2019.

[10] Yaman Umuroglu et al. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[11] Zhang Y. et al. Accurate area, time and power models for fpga-based implementations. In *J Sign Process Syst 63*, pages 39–50, 2011.

[12] Andrea Guerrieri, Srijeet Guha, Chris Lavin, Eddie Hung, Lana Josipović, and Paolo Ienne. Dynarapid: Fast-tracking from c to routed circuits. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 24–32, 2024.

[13] Intel. Quartus Prime Standard Edition User Guide: Design Compilation. https://www.intel.com/content/www/us/en/docs/programmable/683283/18-1/incremental-compilation-for-hierarchical.html. accessed Dec. 2024.

[14] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Invited tutorial: Dynamatic: From c/c++ to dynamically scheduled circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Danielle Tchuinkou Kwadjo and Christophe Bobda. Late breaking results: Automated hardware generation of cnn models on fpgas. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2, 2020.

[16] Danielle Tchuinkou Kwadjo, Joel Mandebi Mbongue, and Christophe Bobda. Exploring a layer-based pre-implemented flow for mapping cnn on fpga. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 116–123, 2021.

[17] Danielle Tchuinkou Kwadjo, Erman Nghonda Tchinda, Joel Mandebi Mbongue, and Christophe Bobda. Accelerating hybrid quantized neural networks on multi-tenant cloud fpga. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 491–498, 2022.

[18] Chris Lavin and Alireza Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140, 2018.

[19] Christopher Lavin, Brent Nelson, and Brad Hutchings. Impact of hard macro size on fpga clock rate and place/route time. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6, 2013.

[20] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. Hmflow: Accelerating fpga compilation with hard macros for rapid prototyping. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 117–124, 2011.

[21] Christopher Lavin, Marc Padilla, Philip Lundrigan, Brent Nelson, and Brad Hutchings. Rapid prototyping tools for fpga designs: Rapidsmith. In *2010 International Conference on Field-Programmable Technology*, pages 353–356, 2010.

[22] Sonia Mami, Younes Lahbib, and Yassine Hachaïchi. Accurate high level resources and power estimators for fpgas. In Med Salim Bouhlel and Stefano Rovetta, editors, *Proceedings of the 8th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT'18), Vol.2*, pages 111–123, Cham, 2020. Springer International Publishing.

[23] Michail Maniatakos, Songhua Xu, and Willard L. Miranker. Constraint-based placement and routing for fpgas using self-organizing maps. In *2008 20th IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 465–469, 2008.

[24] Tommaso Pacini, Emilio Rapuano, and Luca Fanucci. Fpg-ai: A technology-independent framework for the automation of cnn deployment on fpgas. *IEEE Access*, 11:32759–32775, 2023.

[25] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 105:107281, September 2020.

[26] Paul Schumacher and Pradip Jha. Fast and accurate resource estimation of rtl-based designs targeting fpgas. In *2008 International Conference on Field Programmable Logic and Applications*, pages 59–64, 2008.

[27] Frederick Tombs, Alireza Mellat, and Nachiket Kapre. Mocarabe: High-performance time-multiplexed overlays for fpgas. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 115–123, 2021.

[28] M. Xu and F.J. Kurdahi. Area and timing estimation for lookup table based fpgas. In *Proceedings ED&TC European Design and Test Conference*, pages 151–157, 1996.