DeepFlexiHLS: Deep Neural Network Flexible High-Level Synthesis Directive Generator

Mohammad Riazati*, Masoud Daneshtalab*†, Mikael Sjödin*, and Björn Lisper*
*Heterogeneous Systems Research Group, Mälardalen University, Västerås, Sweden
†Department of Computer Systems, Tallinn University of Technology, Tallinn, Estonia
*{mohammad.riazati, masoud.daneshtalab, mikael.sjodin, bjorn.lisper}@mdu.se

2022 IEEE Nordic Circuits and Systems Conference (NorCAS) | 979-8-3503-4550-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/NORCAS57515.2022.9934617

***Abstract*—Deep Neural Networks (DNNs) are now widely adopted to solve various problems ranging from speech recognition to image classification. Since DNNs demand a large amount of processing power, their implementation on hardware, i.e., FPGA or ASIC, has received much attention. High-level synthesis is widely used since it significantly boosts productivity and flexibility and requires minimal hardware knowledge. However, when HLS transforms a C implementation to a Register-Transfer Level one, the high parallelism capability of the FPGA is not well-utilized. HLS tools provide a feature called directives through which designers can guide the tool using some defined C pragma statements to improve performance. Nevertheless, finding appropriate directives is another challenge, which needs considerable expertise and experience. This paper proposes DeepFlexiHLS, a two-stage design space exploration flow to find a set of directives to achieve minimal latency. In the first stage, a partition-based method is used to find the directives corresponding to each partition. Aggregating all these directives leads to minimal latency. Experimental results show 54% more speed-up than similar work on VGG neural network. In the second stage, an estimator is implemented to find the latency and resource utilization of various combinations of the found directives. The results form a Pareto-frontier from which the designer can choose if FPGA resources are limited or are not to be entirely used by the DNN module.**

***Index Terms*—Design Space Exploration, Deep Neural Network, Accelerator, CNN, HLS**

I. Introduction

Deep Neural Networks (DNN) are now used to solve various problems in different domains. They are designed and trained by neural network experts, usually through high-level libraries and APIs like Tensorflow and Keras. After the architecture and parameters are finalized, the network will be ready for the execution phase (inference).

The design and training phase is usually performed on Graphics Processing Units (GPU). However, the inference can be executed on various platforms depending on the application, and the performance is substantially dependent on the execution platform. Traditionally, CPU platforms were used to execute DNN models. Although they did not provide high performance, they were more available and easier to program. GPUs, on the other hand, offer a higher degree of parallelism. In addition to their high power consumption, they have another drawback. Their performance mainly relies on large batch sizes, which does not apply to tasks where the processing latency of a single input is considered, e.g., object detection in real-time applications [1].

The solution that has attracted significant attention for low-latency and real-time applications is accelerating DNNs on platforms like FPGA and ASIC. In addition to their power efficiency, making them more fit for embedded environments, they flexibly provide designers with varied parallelism levels. These platforms allow thousands of different operations, if not interdependent, to be performed simultaneously. The proposed method in this work applies to both FPGA and ASIC platforms. However, we focus on the FPGA and provide our results on it due to its reconfigurability and fast development.

Flexibility in implementing a circuit on an FPGA means that designers can allocate as many hardware resources as they desire, of course, by considering its cost, to get lower latency. Although the synthesis of a DNN on an FPGA seems very attractive, in practice, it faces a serious challenge: neural network design is performed by machine learning specialists using high-level languages, whereas the hardware is implemented by low-level hardware experts. To solve this problem and fill the gap between these two groups, High-Level Synthesis (HLS) tools can be used. HLS receives a high-level design and effortlessly converts it to a hardware design on FPGA.

Despite the emergence of HLS, the problem is still not completely solved. HLS is capable of creating a variety of architectures on the hardware. By default, it creates an architecture with minimal parallelism and resource utilization and, therefore, maximum latency. If the obtained latency is not as desired, users can increase the parallelism level by applying directives. It makes raising the parallelism level effortless. For instance, regarding a loop construct, the user can determine the number of copies of the loop body through the *unroll* directive [2]. Using this directive makes the HLS tool utilize more hardware resources by creating multiple instances of the loop body. Having multiple instances of loop body on the hardware enables the HLS to increase the parallelism, if not limited by other factors like data dependencies.

Applying directives that increase parallelism seems so simple. It might make a designer think of applying all directives to all constructs to achieve maximal parallelism and minimal latency. However, applying the directives does not necessarily lead to a design with added performance in practice. There are cases where the HLS fails synthesizing:

- if the directives applied by the user result in a huge circuit that exceeds the processing capacity of the HLS tool for scheduling
- if the required resources are more than the available resources on the specified chip
- if the concurrent memory accesses needed as the result of applying directives cannot be provided due to the limited ports of the specified memory unit

Another challenge in determining a set of directives for a specific design is the extra resource utilization caused by applying it. Increasing the parallelism usually comes at the cost of utilizing more hardware resources. However, two sets of directives with the same performance effect might require a different amount of resources.

For a small design, setting HLS directives is relatively simple and can be handled manually, even through a trial-and-error procedure. However, in the case of DNNs, it is significantly challenging. DNNs usually consist of many loops. Besides, most of the loops are deeply nested meaning that a directive on one loop can affect another directive on another loop, either an inner or outer one. Therefore, determining the proper directives is a problem with a vast search space. This paper proposes DeepFlexiHLS as an automated two-stage design space exploration (DSE) method for determining the appropriate HLS directives for DNNs.

In summary, the main contributions of the proposed method are as follows:

- An effortless and automatic flow from a high-level description of a DNN to a high-performance hardware implementation is provided.
- A set of directives to achieve minimal latency is found (stage one).
- An estimator to find an HLS result without running it is configured and tested. Then, the synthesis results for many possible directive sets are estimated, resulting in a Pareto-frontier of their achieved latency and resource utilization to enable designers to choose according to their requirements and resource availabilities (stage two).

DeepFlexiHLS is tested on two well-known networks, LeNet [3] and VGG [4], and the results are presented and compared with similar recent previous work.

The rest of this paper is organized as follows. In Section II, we review some related work on DNN acceleration. The proposed method is elaborated in Section III, and the experimental results are provided in Section IV. Then, Section V discusses some possible challenges. Finally, Section VI concludes the paper.

II. Related work

The acceleration of DNNs on FPGA has recently received much attention and has been covered in numerous articles. Related work of the present article can be considered in different categories, which we cover in this section.

A group of methods directly generate RTL code that can be synthesized on the FPGA. Some, such as [5] and [6], introduce a customized accelerator for the specific network presented in their articles, to classify MNIST and ImageNet images respectively. In [7], the hardware architecture of a computing unit is proposed, and the steps to implement a convolutional neural network using configured units are explained. In [8], hand-optimized design templates are used to generate a synthesizable code for the accelerator.

Another group is methods based on heterogeneous systems. In these methods, predefined operations are designed and implemented as hardware elements on the FPGA and are controlled and scheduled on a CPU. They may use OpenCL [9], [10], Legup Pthread [11] or heterogeneous synthesis tools such as Xilinx SDAccell or Xilinx SDSoC [12], [13].

Both of these groups of methods require neural network designers to be familiar with hardware details such as memory structure or component communication methods. They are relatively complex for them to use [10]. Besides, they suffer from the problem of inflexibility, i.e., the user must accept a certain delay and utilization point. A user cannot sacrifice delay for utilization or vice versa, considering their requirements and FPGA resources.

Some others use HLS to synthesize DNNs on FPGAs. HLS alone cannot provide adequate performance and only transforms the high-level implementation (mostly in C, C++, or SystemC) to a register-transfer level one (mostly in VHDL or Verilog). Thus, each of these works strives to improve the results through their proposed methods. [14] suggests using activation functions with customized implementation. [15] implements a specific DNN and synthesizes it using the HLS. It manually adds directives to that network to improve its performance. In [16], unroll and pipeline directives are used to improve loop parallelism. It puts these directives in specific and predefined locations and does not consider the properties of each individual DNN. In [17], an LLVM-based approach is used to find appropriate directives. We have adopted an approach similar to this group. We propose a generic method that applies to any DNN. No predefined directive locations exist, and different results are provided to enable the user to choose a desirable latency-utilization point.

There are also some works that, although they are not still applied to the DNNs, can be considered related to the present work. Those are methods that suggest DSE to find appropriate HLS directives for a high-level design, not necessarily DNNs. They propose heuristic methods to explore the possible directives for the HLS. Some of them, such as [18], propose the division of a design and perform a complete DSE on each part. None of these methods can be used for DNNs with deeply nested loop structures (e.g., six levels). The directives applied to the nested loops are not independent and affect each other's results. Besides, the full exploration, or even exploration using methods such as Genetic Algorithm (GA), will fail due to the extremely large number of possibilities, e.g., billions of combinations for a single convolutional layer. Some others, such as [19] and [20], use machine learning methods. These methods fail, too, for this gigantic search space. Therefore, we propose a directed search method to ensure that the complexities of a DNN implementation are taken into account.

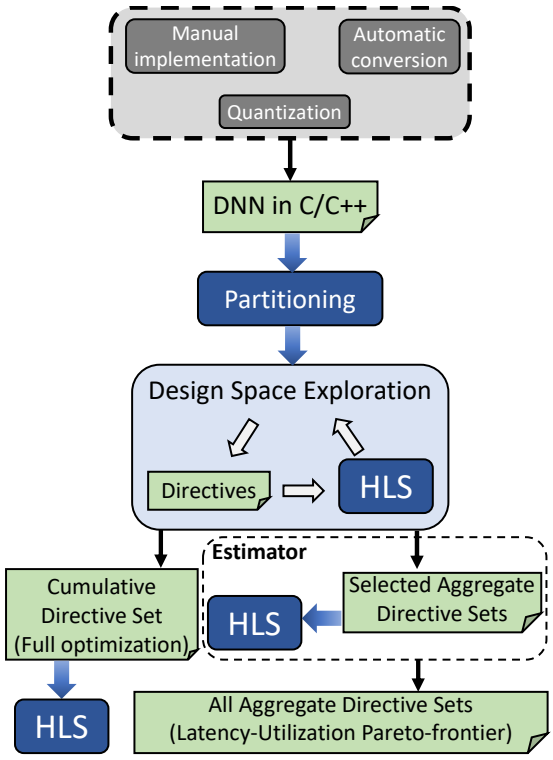

Fig. 1. The overall view of DeepFlexiHLS.

III. Proposed Exploration Method

The primary input of the proposed method is the DNN implementation. This source is then used both in the exploration and the HLS runs. In the end, a directive set (called a cumulative directive set) is created that results in minimal latency. Besides, a set of Pareto-optimal points are created to help designers choose a latency-utilization point based on their requirements and resource limits. Figure 1 shows an overview of DeepFlexiHLS.

The input DNN implementation must be provided in C. It can be created manually by an expert familiar with the functionality of network layers or automatically, using the tools to convert high-level DNN descriptions into C. It should be noted that the conversion tool should be HLS aware in order to generate synthesizable C implementation, e.g., [21]–[23]. We used DeepHLS tool introduced in [22]. This tool receives the implementation of a DNN in Keras [24] and generates a C code to be synthesized by HLS and a corresponding testbench to test the C implementation.

A. Partitioning

A DNN implementation in C consists of a large number of loop statements, and various HLS directives can be applied to each of them. This makes the exploration space extremely large. One solution is partitioning the design into less independent sections and exploring each section separately. Each loop, together with

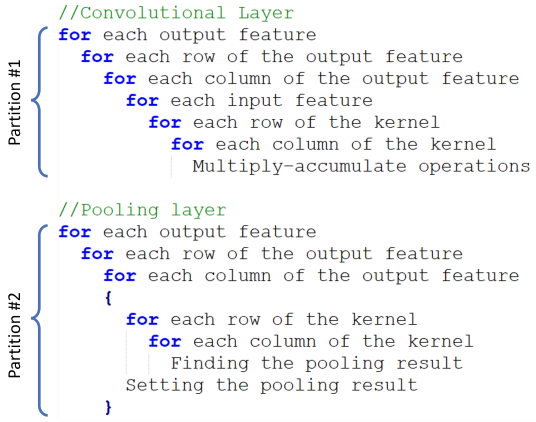

Fig. 2. A pseudo code of partitioning two example DNN layers.

its inner nested loops, is a proper candidate for being a separate partition; the loops in each partition significantly interact to calculate the values for the output matrices, and two consecutive partitions only depend on the output matrix of the first one of them.

In order to create the list of partitions, we first determine the parent loop for each loop. A loop without a parent together with all the loops embodied in it is tagged as a separate partition.

It should be noted that the partitioning does not change the C code but configures the DSE to perform the exploration on each partition separately and find the corresponding directives. In other words, every run of the HLS is executed on the entire C implementation, including all partitions, while the corresponding directives of only one partition are applied. Figure 2, as an example, shows the result of partitioning two popular DNN layers (a convolutional layer and a pooling layer).

B. Design Space Exploration

Now that the C implementation is ready and partitioned, the first stage of the exploration can begin. The objective of the first stage is to explore a set of HLS directives to achieve minimal latency. Another possible objective (or constraint) can be resource utilization which is covered in the second stage (Section III-C).

A DNN implementation consists of several layers. The number of loops may vary depending on the layer type. For example, convolutional, pooling, or fully connected layers consist of 6, 5, and 2 loops. It is possible for each of the loops to pipeline, unroll, or both. When the pipeline directive is applied to a specific loop, the next iteration of the loop is executed as soon as the required data is ready and does not wait for the current iteration to finish. By applying the unroll directive, instead of implementing only one copy of the loop content and using it for all iterations, several copies of the loop content are created to run in parallel.

Each loop can be pipelined or not. However, in the case of the unroll directive, the number of possibilities is much higher. In fact, for a loop with n iterations, any unroll factor between two and n is possible. After several tests on layers of various sizes, we noticed that most of the time, the lowest latency occurs on one of the divisors of n. Besides, when the unroll factor is one of the divisors, the HLS will not need to implement the exit condition test between iterations. Therefore, in the proposed method, we consider only the divisors of the n. Note that, regarding pipeline directives, there is an extra parameter known as Initiation Interval (II). In this work, we always used the default value (one).

Table I lists possible parameters for each for-loop of a small example convolutional layer. For each row, the number of possible directive sets is mentioned in the last column. For example, for the first row, there are two possible options for the pipeline, and seven for the unroll, which results in 14 possible combinations. Note that an unroll factor of one denotes not applying it.

It is noteworthy that there are more directives that affect the performance of the implementation. In addition to the unroll and pipeline directives that are automatically inserted, we manually inserted array-partitioning directives. Automating the insertion of them and considering some other directives is considered future work. There are also some directives that are irrelevant. For example, some HLS tools support function pipelining, which could not be used in this work since the design is fully flat (i.e., only containing for-loops but no functions). It should also be noted that, in addition to the directives, there are some global HLS run configurations, such as target device and clock frequency, that were manually set and presented in the experimental setup.

TABLE I
POSSIBLE DIRECTIVES FOR AN EXAMPLE CONVOLUTIONAL LAYER

Nesting level	Iterations	Pipeline options	Unroll options	Directive set possibilities
1	64	No, Yes	1, 2, 4, 8, 16, 32, 64	14
2	16	No, Yes	1, 2, 4, 8, 16	10
3	16	No, Yes	1, 2, 4, 8, 16	10
4	64	No, Yes	1, 2, 4, 8, 16, 32, 64	14
5	5	No, Yes	1, 5	4
6	5	No, Yes	1, 5	4

By multiplying the "possibilities" values, it is concluded that more than 300 thousand combinations exist for a single layer ($14\times10\times10\times14\times4\times4$). It means that neither an exhaustive nor a random algorithm, like the Genetic Algorithm (GA), can handle such a vast search space.

In this work, we adopted a controlled algorithm to traverse through possible combinations. The algorithm begins with choosing a single directive among all possible directives and continues by gradually increasing the number of active directives and forming a directive set. Algorithm 1 lists the steps of the DSE execution procedure.

The algorithm begins with running HLS on the base design without including any pipeline or unroll directives. This is equivalent to setting the pipeline options to "No" and unroll options to one for all of the for-loops. The resulting report of the HLS is then parsed and analyzed to extract the latency of the resulting circuit. Then, the possible directive sets for each partition are explored. The following paragraph explains how DSE is performed on a specific partition.

Starting from a current directive set, which in the beginning is an empty set, a group of neighbor sets are created. A neighbor set is formed by adding just one directive to the current set. Note that all the neighbors are within the partition under process. The result of the HLS for each of these neighbors is obtained. Then, the one with minimum latency will be chosen as the winner of the current round. The iteration continues until no improvement is made by adding directives. The last directive set is stored for the current partition and will be used later to form directive sets for the entire design.

In order to obtain the latency when a directive set is applied, HLS is needed to be executed. HLS is considered a black box that synthesizes a design according to specified directives and then reports the latency and resource utilization. In order to save on the number of HLS executions, which is the most time-consuming part of the algorithm, the following facts are considered. They lead to finding the result of applying a directive set without executing the HLS. In fact, before running the HLS for a specific directive set, the directive set is simplified. If one of the previous directive sets matches this simplified one, its result is used without running the HLS.

- If a loop is pipelined, HLS completely unrolls all the loops inside it (at any level). Therefore, unroll directives on the inner loops are all ignored by HLS.
- When a loop is completely unrolled, pipelining it will be ineffective and ignored by HLS.
- If a for loop has only one iteration, e.g., in implementing a convolutional layer with one input feature map, pipelining and unrolling are irrelevant and ignored by HLS.

Due to the particular structure of the DNN implementation, these simplifications result in a significant saving in the number of the HLS runs. For instance, 32.76% fewer HLS runs were required for the exploration of the LeNet network.

As mentioned earlier, after the DSE for each partition is performed, the winning directive set is stored for the following stages. However, at this point, the first result of the flow can be produced: Cumulative Directive Set (CDS). CDS is created by combining all the directive sets. It includes all the directives related to all partitions and results in the minimal latency of the design (Full optimization in Figure 1).

The problem that might occur is that the result might not fit on the specified target FPGA if the neural network under process is very large. Improving the latency is achieved through increasing the parallelism, and parallelism comes at the cost of extra resource utilization. Therefore, it is possible that the extra required resources are not available on the target chip or are not supposed to be used due to some considerations. In the second stage of the DSE, we will present a method that finds an aggregate directive set that can fulfill the resource constraints.

Algorithm 1: Partitioned DSE Algorithm (Stage 1)

I: Initial directive set (an empty set)
LI: the latency if directive set I is applied
DS: a Directive Set
SDS: a Simplified Directive Set
PML$_i$: the minimum found latency for a partition
PDS$_i$: the directive set corresponding to a PML$_i$
neighbor$_j$ **of PDS**$_i$: a DS which contains only directives of partition$_i$ and has one more active directive than PDS$_i$
Simplify(X): The simplified version of the directive set X
HLS(X): Runs HLS while applying the directive set X and returns the Latency
Output: PDS$_i$ for each partition$_i$.

```
LI ← HLS(I);
foreach partition_i do
    MinLatency ← LI;
    DS ← I;
    repeat
        PML_i ← MinLatency;
        PDS_i ← DS;
        foreach neighbor_j of PDS_i in partition_i do
            SDS ← Simplify(neighbor_j);
            if HLS is executed on SDS before then
                Latency for SDS ← existing result of SDS;
            else
                Latency for SDS ← HLS(SDS);
        MinLatency ← Min(Latencies of all SDSs) ;
        DS ← neighbor_j which resulted in MinLatency;
    until PML_i < MinLatency
```

C. Handling resource constraints

In the previous stage, the required directives to improve the latency of each partition were found. If all the available resources of the FPGA are to be consumed by the implemented DNN, all the found directive sets of all the partitions will be accumulated and applied to the implementation. However, if some FPGA resources are needed for other system components or a smaller FPGA is going to be used, then the designer might decide not to apply the found directives of all the partitions but a subset of them. In that case, an important question will be raised: which subset of the found directives should be applied to achieve minimal latency while considering the constraints?

Each combination of selected partitions results in a different latency and resource overhead. The number of combinations might be significant, e.g., 1,048,576, for a twenty-partition design, equal to the number of possible subsets of a set of size 20. It is not practical to collect the HLS result for all these combinations This section implements an estimator to find the latency and resource utilization if a specific directive set is applied without running HLS. Then, the estimator is used to find the latency and resource utilization for each of the combinations. Note that this stage solves neither an optimization nor a constraint satisfaction problem. Instead, it finds the latency and resource utilization of all possible combinations with the help of an estimator and provides the designer with the result (as a Pareto-frontier) to choose the desired latency-utilization point.

An Aggregate Directive Set (ADS) for a combination of partitions is created by merging all the directives found for each of the partitions. Applying the found directive set for Partition$_i$ will cause L$_i$ decrease in the overall latency and R$_i$ increase in the utilization of a specific resource. Through examining the synthesis result for various ADSs, we observed that, as was expected because of the structure of the C code of DNNs, the total decrease in latency and the total increase in resource utilization for an ADS can be roughly calculated as the sum of all L$_i$s and R$_i$s of all included partition in that ADS. However, there was an exception to that. The synthesis result could differ more if there were adjacent partitions included in an ADS. In this case, HLS could perform more optimizations. Therefore, in order to have a more accurate estimator, it is also beneficial to collect the HLS results for some additional ADSs (Selected ADSs in Figure 1).

An Adjacent Partitions List (APL) is a set of partitions that appear consecutively in the design. For instance, assuming a design with four partitions $(1, 2, 3, 4)$, there can be two APLs of size 3: $\{1, 2, 3\}$, $\{2, 3, 4\}$. Formally, APL$_{i,j}$ is defined as an APL starting from partition i having a size of j. Accordingly, APL$_{2,3}$ $\equiv \{2, 3, 4\}$. For example, for a twenty-partition design, there are 19, 18, and 2 APLs that contain two, three, and 19 consecutive partitions, respectively. Therefore, for a twenty-partition design, another 189 $(19 + 18 + ... + 3 + 2)$ Selected ADS runs (out of 1,048,576 possible combinations) are needed. In the experimental results section, it will be shown how this enhances the estimator's accuracy.

An ADS for a Partitions List (PL), noted as ADS(PL), is created by merging all the directives found for each of the partitions in PL. Note that, unlike an APL, the partitions in a PL are not necessarily adjacent.

The estimator calculates the latency and utilization of the implementation when ADS(PL) is applied by finding the set of largest APLs included in the PL, called LAPL (Largest APL List). Formally:

$$\text{LAPL(PL)} = \{\text{APL}_{i,j} \mid \nexists\, x \text{ such that } \text{APL}_{x,j+1} \subseteq \text{PL}\} \tag{1}$$

For example, in the case that PL = $\{1, 2, 4, 5, 6\}$, in which partition 3 is not in the PL, then LAPL(PL) = $\{\{1, 2\}, \{4, 5, 6\}\}$.

Now, LAPL(PL), which is the list of the APLs included in PL, is ready, and the estimated effect of PL on latency and utilization can be calculated. EOL(ADS)/EOU(ADS) is the Effect On the Latency/Utilization of the implementation as a result of applying ADS. The effect is the amount of decrease/increase when compared with a design with no directives. Note that EOL(x) and EOU(x) in the following relations are already found through executing HLS on Selected ADSs, and no more HLS execution will be needed.

$$\text{EOL(PL)} = \sum_{x \in \text{APPL(PL)}} \text{EOL(x)} \tag{2}$$

$$\text{EOU(PL)} = \sum_{x \in \text{APPL(PL)}} \text{EOU(x)} \tag{3}$$

The final step will be the calculation of the latency and utilization. For a specific PL, the latency/utilization is calculated by adding EOL(PL)/EOU(PL) to the latency/utilization of the base implementation. The base implementation is the HLS result when no directive is supplied.

There are some points to be considered before concluding this section.

- The order of complexity of the estimator algorithm is O(n^2), in which n is the number of partitions in the design (i.e., the number of network layers).

- In this section, the term (resource) utilization was generically used for any of the FPGA resources, including DSP, LUT, or FF.
- The proposed exploration method has no effect on the accuracy of the DNN since the source design, i.e., the C implementation, is fixed all through the process.
- Some HLS tools also support directives such as dataflow, which greedily analyze the implementation (after applying other directives) and perform some more latency optimizations. In this work, we have not considered these directives. If they are also to be added, then the proposed estimation method needs modifications.

IV. Experimental Results

In order to demonstrate the effectiveness of DeepFlexiHLS, we first compare its results with previous work. Then we investigate how the proposed estimator and its result Pareto-frontier are formed to handle resource constraints.

The first stage is to create a synthesizable C implementation for the DNN in question. For this, we used the tool presented in [22]. There are many options to specify for the tool before generating the C implementation, e.g., storage location for network parameters, i.e., BRAM (default) or external RAM, order of the for-loops in each layer, and label formats for the for-loops. For all of them, we chose the default settings. Note that it is just about the input design, and the exploration flow will not be affected.

A. Full optimization results

When all the found directives for all partitions are accumulated and applied to the design, the full optimization result is achieved. We examined our proposed method on both a small and a large network: LeNet [4] and VGG [3]. These convolutional networks are frequently used in the literature to evaluate various methods.

We found two previous works using HLS to synthesize LeNet. [15], [16]. Both of them apply pipeline and unroll directives to some specific, predetermined loops. The obtained latency for each of them is 410,758 and 913,516 clock cycles, respectively. In contrast, DeepFlexiHLS reaches 902 clock cycles.

In a recent work [17], a similar directive-based HLS optimization method is adopted. It uses an LLVM-based approach, connects various third-party tools to create a C code, and finally finds the appropriate directives. To the best of our knowledge, it is the only work that uses pure HLS to implement a large network, such as VGG. We used the same experimental setup as theirs: Xilinx Vivado HLS, Xilinx VU9P FPGA, the clock frequency of 200 MHz, and eight-bit quantization. In order to create the C implementation of VGG, we used the method proposed in [22]. As demonstrated in Table II, the proposed method provides 54% improvement in the latency. Although the DSE enabled us to increase the parallelism and therefore decrease the latency, higher parallelism comes at the cost of more resource utilization. Therefore, as we expected, the resource utilization of our implementation, i.e., DSP and LUT, is higher.

TABLE II
Optimization result comparison with [17]

	[17]	DeepFlexiHLS
Speedup	1505.3x	2325.6x
DSP Utilization	38.50%	49.65%
LUT Utilization	22.40%	31.75%
FF Utilization	Not Reported	25.76%
BRAM Utilization	Not Reported	3.38%

B. Configuring the estimator

As explained in section III-C, in order to handle resource constraints, we need to have the synthesis results of various aggregate directive sets (ADS). An ADS consists of the found directives of a selected set of layers. For instance, for a ten-layer DNN, 2^{10} different ADSs exist. Each execution of HLS can take from a few minutes to a few hours. Therefore, collecting the HLS results for all of them by running HLS is impractical. To find the HLS results of all possibilities, we implemented the estimator discussed in III-C, including adjacent partitions. As stated earlier, the reason for collecting the HLS results for adjacent partitions is that when directives are added to two (or more) consecutive layers, HLS applies some additional improvements. Figure 3 shows some randomly selected ADSs to test the accuracy of the estimator. For each of them, HLS was executed (the blue curve) to check how accurate the estimator result is. As can be seen, the configuration that uses consecutive partitions as the source of estimation, i.e., the black curve, is closer to the actual HLS results and overlaps the blue curve most of the time. Through this, the average accuracy error of the latency estimator decreased from 9.74% to 3.86%. A similar result was obtained for the accuracy of the LUT and FF utilization estimator: 8.31% and 6.45% improvement, respectively.

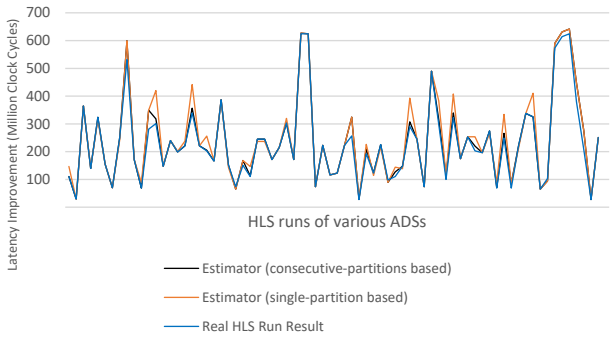

Fig. 3. Configuring the estimator.

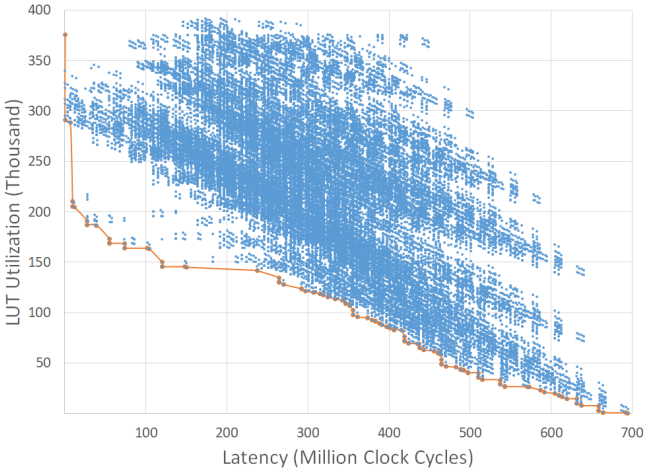

Fig. 4. Pareto-frontier of all aggregate directive sets.

C. Latency versus resource utilization Pareto frontier

Using the configured estimator, the latency and resource utilization of various ADSs can be obtained. Obviously, the two objectives are contradictory. Therefore, the (full) exploration has no optimal point but forms a Pareto frontier. Figure 4 shows the latency and LUT utilization of different ADSs. The maximum latency is for the ADS that utilizes the minimum number of LUTs, and vice versa. Each point is associated with an ADS and specifies the partitions that their related directives should be applied. The points on the Pareto frontier, the orange curve, are the points that are reasonable to be chosen by a designer considering the acceptable latency and the resource constraints.

V. Discussions

While implementing the DSE, we considered the HLS as a black box. It means that we neither had access to nor needed the internal data structure of the HLS. However, there are some points that a change in the HLS can affect our DSE implementation,

not its overall flow. Firstly, supplying the generated directive set might differ between various HLS tools. It is usually possible through the directive files or the pragma directives. Directive files are separate files supplied to the HLS along with the main design. The directives are assigned to each language construct, e.g., for-loops, using their labels. It means adding the labels to the source files, though they might be optional for executions on other platforms, is mandatory. Another alternative to supply the directives is through some pragmas embedded in the source code just below or above a specific language construct. Some HLS tools might support one or both of these alternatives. Besides, there is no specific standard for these directives' syntax, and each HLS tool uses its own defined format. Another point is the generated report. Each HLS tool provides the report in its specific format. It means that the DSE module responsible for reading the latency from the generated report might be different from one HLS tool to another.

Another aspect of the proposed method that might be argued is its execution time. There are many combinations of directives that are expected to work but do not due to some internal HLS processing limits. Besides, even if a synthesis can run successfully, the result might be different from the expectation. Running HLS for the exploration has an important advantage: the exploration is performed based on the real behavior of the tool. However, it comes at the cost of exploration time. In this work, there were many HLS runs that could be performed in parallel, and thanks to a server with multiple CPUs, we managed to collect the results much faster. However, we still admit that the method will take some time and should probably be performed at the final stages of a design lifetime.

The last thing worth mentioning is the simplicity of using the tool, e.g., the learning curve of using the toolchain or the setup time. Considering the automatic generation of C implementation, e.g., through using [22], the whole flow will be automatic and minimal setup is required to collect the result. The most important settings are the target FPGA, the expected FPGA logic frequency, and settings related to the C code generation.

VI. Conclusion and future work

In this paper, we proposed DeepFlexiHLS, a partition-based DSE method to explore possible HLS directives to improve the performance of running the DNN on the FPGA. The main advantage of using HLS to implement a DNN is being straightforward and easy to use for high-level DNN designers. Besides, it allows designers simply switch from one implementation to another based on the requirements and constraints. Its primary disadvantage is that the generated circuit may have higher latency than other methods. In this work, we targeted and solved this issue. The method was tested on the VGG network, which is among the largest DNNs benchmarked in the literature. The proposed method first found the design with minimal latency disregarding the resource utilization overhead, and then, through configuring an estimator, explored all possible combinations of layer optimization directives to provide a Pareto-frontier of latency and resource utilization.

We considered pipeline and unroll directives in the present work. Most HLS tools support more directives, especially to tweak the memory access performance, e.g., through array partitioning. In our future work, we are going to analyze and automate the insertion of more HLS directives.

Acknowledgment

The work presented in this paper was supported by the Swedish Knowledge Foundation via HERO project. It has been also partially conducted in the project ICT programme which was supported by the EU through the European social fund.

References

[1] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 37, no. 1, pp. 35–47, 2017.

[2] Xilinx, "Vivado design suite user guide: High-level synthesis," 2019.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] R. G. S. Ram, N. Chaturvedi, S. Saurav, and S. Singh, "An fpga based hardware accelerator for classification of handwritten digits," in *International Conference on Intelligent Systems Design and Applications*. Springer, 2018, pp. 945–954.

[6] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[7] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 2, pp. 326–342, 2018.

[8] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[9] A. Ghaffari and Y. Savaria, "Cnn2gate: Toward designing a general framework for implementation of convolutional neural networks on fpga," *arXiv preprint arXiv:2004.04641*, 2020.

[10] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 152–159.

[11] J. H. Kim, B. Grady, R. Lian, J. Brothers, and J. H. Anderson, "Fpga-based cnn inference accelerator synthesized from multi-threaded c software," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2017, pp. 268–273.

[12] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[13] P. G. Mousouliotis and L. P. Petrou, "Cnn-grinder: From algorithmic to high-level synthesis descriptions of cnns for low-end-low-cost fpga socs," *Microprocessors and Microsystems*, vol. 73, p. 102990, 2020.

[14] S. Ghaffari and S. Sharifian, "Fpga-based convolutional neural network accelerator design using high level synthesize," in *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*. IEEE, 2016, pp. 1–6.

[15] D. Rongshi and T. Yongming, "Accelerator implementation of lenet-5 convolution neural network based on fpga with hls," in *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*. IEEE, 2019, pp. 64–67.

[16] M. Cho and Y. Kim, "Implementation of data-optimized fpga-based accelerator for convolutional neural network," in *2020 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE, 2020, pp. 1–2.

[17] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[18] L. Ferretti, G. Ansaloni, and L. Pozzi, "Cluster-based heuristic for high level synthesis design space exploration," *IEEE Transactions on Emerging Topics in Computing*, 2018.

[19] N. Wu, Y. Xie, and C. Hao, "Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 39–44.

[20] M. Yu, S. Huang, and D. Chen, "Chimera: A hybrid machine learning-driven multi-objective design space exploration tool for fpga high-level synthesis," in *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 2021, pp. 524–536.

[21] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[22] M. Riazati, M. Daneshtalab, M. Sjödin, and B. Lisper, "Deephls: A complete toolchain for automatic synthesis of deep neural networks to fpga," in *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2020, pp. 1–4.

[23] M. Riazati, M. Daneshtalab, M. Sjödin, and b. Lisper, "Autodeephls: Deep neural network high-level synthesis using fixed-point precision," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2022, pp. 122–125.

[24] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.