Sensei: An Area-Reduction Advisor for FPGA High-Level Synthesis

Hsuan Hsiao, Jason H. Anderson
Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada
julie.hsiao@mail.utoronto.ca, janders@ece.utoronto.ca

***Abstract*—High-level synthesis (HLS) provides an easy-to-use abstraction for designing hardware circuits. However, standard datatypes in high-level languages are overprovisioned for typical applications, incurring extra area since the underlying FPGA hardware can support arbitrary bitwidths. This area inefficiency can be overcome by enabling the use of arbitrary-width datatypes at the source code level. However, this requires that HLS users spend time and effort on examining all program variables and quantifying their area impact, which can be intractable especially with large, complex programs and time-consuming synthesis. We propose *Sensei*, an advisor that predicts the post-synthesis area savings brought about by reducing bitwidth and presents users with a ranking of program variables and their area impact. Equipped with a convolutional neural network (CNN)-based predictor, Sensei achieves high area-prediction accuracy and enables rapid exploration of area-saving opportunities.**

I. Introduction

High-level synthesis (HLS) eases hardware design and verification by allowing such tasks to proceed at a higher level of abstraction. HLS has become a mainstream design methodology for field-programmable gate arrays (FPGAs), with both Xilinx and Altera/Intel offering commercial HLS solutions. The rising importance of HLS is underscored by the fact that the vendors themselves now use their own HLS solutions to produce compute accelerators for specific application domains, such as machine learning [1], [16]. A key HLS research challenge is tied to the "black box" nature of HLS and the consequent disconnect between the software program (input) and the generated circuit (output). This disconnect significantly impedes HLS usability and leads to the following key question: how should a software developer *change* their program to produce a better quality HLS-generated circuit? In this work, we take a step towards answering this question, with emphasis on the circuit area quality metric.

A key advantage of using custom hardware vs. a microprocessor is that datapath widths can be tailored in hardware to match the application at hand; whereas in a processor, widths are fixed at typically 32 or 64 bits. A weakness of the HLS methodology is that the languages used to specify the input (such as `C`) have types that are quantized in their widths, typically 8, 16, 32 and 64 bits. This makes it challenging for HLS users to produce circuits with custom-width datapaths. To bridge the gap between the overprovisioned datatypes of high-level languages and the arbitrary-width datapaths of low-level circuits, static compiler analyses have been developed to *automatically* infer the use of narrower widths, providing an average of 18% area savings [4], [5]. However, compiler passes are conservative, primarily because they execute without knowledge of program input data. To combat this, HLS tools, such as Xilinx's Vivado HLS, allow the user to specify non-standard bitwidths for variables that are known to not require the full width of a standard type. This yields higher area savings (e.g. 54% area savings in an ideal scenario where all input values are known a priori [5]), but is time-consuming and difficult since there may be many variables and the relationship between a program variable's width and final circuit area is somewhat opaque.

In this paper, we propose *Sensei*, an HLS advisor that estimates the area reductions to be gained by bitwidth reductions in program variables. It leverages the availability of both static compiler bitwidth analysis and arbitrary-bitwidth types in the HLS tool. Sensei provides the programmer with a list of key variables to focus on for bitwidth reduction; the reductions in the widths of these variables are likely to have the most impact on final circuit area.[1] Area reduction predictions are made using a convolutional neural network (CNN) that we train solely using information available at the HLS compiler stage, enabling portability across different hardware platforms.

The contributions of this paper are as follows:

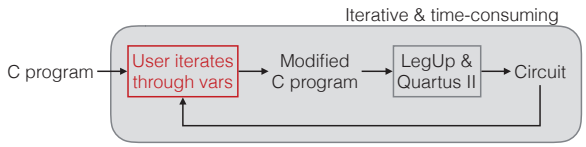
- *Sensei*: an area-reduction advisor for HLS that uncovers opportunities for area savings through bitwidth reduction of program variables. Upon examining the report generated by Sensei, the user is equipped with the knowledge of *which* variable can provide the greatest area savings and *how much* area savings to expect.
- We explore the use of CNNs to predict the area impact of source-code bitwidth changes. To the authors' knowledge, this application of CNNs is the first of its kind. We introduce a novel transformation of the compiler's dataflow graph (DFG) and HLS metadata into a spatial representation well-suited to CNNs.
- We implement Sensei on top of the LegUp HLS tool [2]. We perform an experimental study to evaluate a range of CNN models and demonstrate their efficacy compared against a traditional analytical approach.

II. Background

We review background needed to understand the Sensei framework: HLS and bitwidth-driven area reduction in HLS, the LLVM compiler, and CNNs.

[1]This list indicates the area savings potential of each variable. Sensei assumes that the users will only reduce a variable's width when it is appropriate (and correct) in their application.

978-3-9819263-0-9/DATE18/©2018 EDAA



(a) DSE with conventional iterative approach

(b) DSE with Sensei

Fig. 1: Design space exploration steps with (a) a conventional iterative approach; and (b) Sensei.

A. HLS and Bitwidth Optimization

The traditional steps in HLS include *allocation*, *scheduling*, *binding* and *RTL code generation* [3]. HLS tools typically introduce several HLS-specific optimizations in order to generate higher quality circuits. One such optimization that makes up an integral part of our approach is *bitwidth minimization*. Software programming languages such as `C` use predefined types, typically 8, 16, 32 or 64 bits wide. To reduce circuit area consumption, LegUp HLS incorporates bitwidth minimization [5] to shrink bitwidths at compile time to widths below those specified by the programmer. Generally, this works by statically analyzing and propagating the impact of constants in the source code. For example, consider the logical AND: `z = a & 0x1F`, where `z` and `a` are declared as unsigned 32-bit integers. In this case, `z`'s bitwidth can be reduced to 5 bits, as its upper-most bits are guaranteed to be 0. Moreover, this can be propagated to `a`, whose bitwidth may be reduced to 5 bits, depending on whether it is used elsewhere in the program. This propagation is performed on all instructions in the program, resulting in bitwidth reductions on other variables in the DFG. The interested reader is referred to [5] for complete details.

B. LLVM Compiler

The open-source LegUp HLS tool we use to evaluate our work is built as a back-end to the LLVM compiler framework [9]. Within LLVM, the program being synthesized is represented in an *intermediate representation* (IR) – machine-independent assembly code comprising basic arithmetic and logical instructions (e.g. `add`, `xor`, `mul`) and control-flow instructions (e.g. `br`, `call`). From the IR, it is straightforward to construct a *dataflow graph*, where the nodes in the graph represent the program's arithmetic/logical operations, and edges represent data dependencies. A key contribution of our work is an approach for representing a DFG in a manner suitable for input to a CNN.

C. CNNs

Applications of CNNs have exploded in recent years, and CNNs are delivering state-of-the-art results in image recognition (e.g. [14]), game playing, and other domains. While a detailed overview of CNNs is beyond the scope of this paper (see [13]), we briefly review key concepts. A CNN comprises a multi-layered network of neurons, with weighted edges connecting neurons in different layers. As in traditional neural networks, CNNs contain fully connected layers where neurons have weighted connectivity to *all* neurons in the preceding layer. However, with CNNs, most computational work resides in convolutional layers, where neurons have weighted connections to spatially localized subsets of neurons in the preceding layer. This allows for efficient identification of local features in image-like inputs.

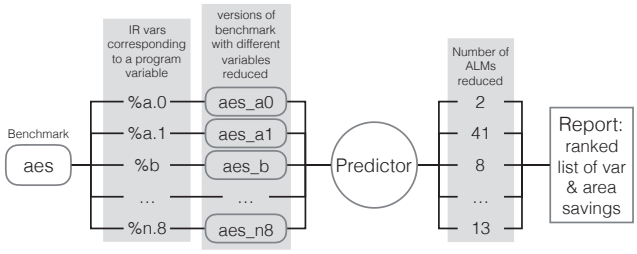


Fig. 2: Example of Sensei ranking the area impact of variables in a benchmark named `aes`.

III. Overview of Approach

Sensei provides designers with design-space exploration (DSE) assistance, allowing them to focus their effort on changes likely to yield appreciable area savings. Fig. 1a depicts the normal DSE steps for tuning an HLS-generated circuit using program variable bitwidths. The designer constrains the width of `C` program variables, synthesizes the circuit to obtain the area impact, and selects the next variable(s) to constrain. The normal DSE approach is time-consuming as full synthesis is executed multiple times, and moreover, it relies heavily on designer intuition and effort to decide what to try next. Contrast traditional DSE with the proposed advisor-based approach, shown in Fig. 1b, where Sensei accepts the original `C` source code as input and generates a ranked list of variables and their corresponding area impact. Instead of having to manually iterate through all variables to find the highest potential for area reduction, the designer can focus their effort on exploring only those variables most likely to yield area reductions. The report generated by Sensei also provides an overall picture of how much area savings can be achieved by bitwidth reductions of specific variables. The designers can optionally present a modified bitwidth-reduced program to Sensei to obtain a new list of variables if they wish to squeeze out more area by examining whether any new variables will become an effective source of area reduction after a first batch of changes. This flow removes full synthesis from the design iterations, enabling the designer to explore a wide range of designs rapidly.

IV. Sensei

Fig. 2 depicts how Sensei ranks the area impact of a program's variables. For a given `C` program, Sensei automatically enumerates all IR variables that correspond to `C` program variables. For each IR variable, it generates a *hypothetical* version of the original program's DFG with the IR variable's

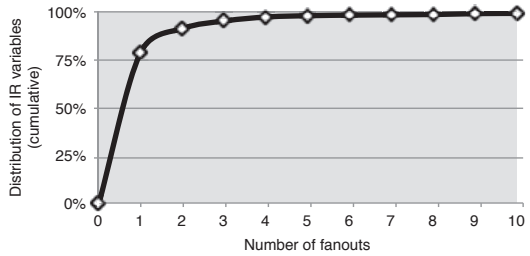


Fig. 3: Average distribution of the number of fanouts for each IR variable.

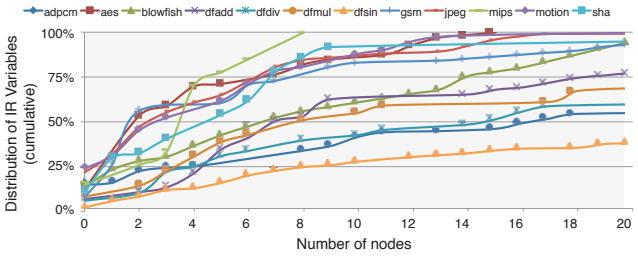


Fig. 4: Cumulative distribution of the depth of nodes above a node that corresponds to a program variable.

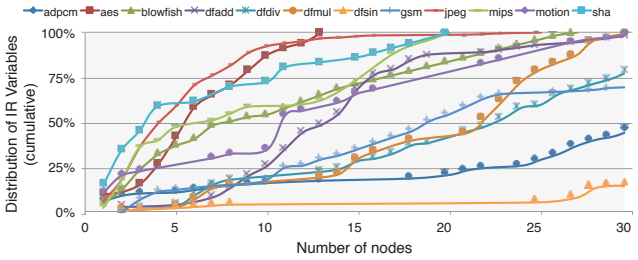


Fig. 5: Cumulative distribution of the depth of nodes below a node that correspond to a program variable.

width reduced by a fixed pre-defined amount. Then, LegUp's bitwidth minimization framework is executed to propagate the hypothetical bitwidth reduction through the DFG, thereby allowing bitwidths of other variables to also be reduced. The area-reduction predictor in the advisor then generates a prediction of the post-synthesis area reduction afforded by the hypothetical bitwidth-reduced DFG. In the end, the advisor gathers its predictions for all variables and generates a ranked list of variables and their area impact.

The key technical challenge is in predicting the post-synthesis area reductions associated with a candidate variable bitwidth change. For this, we use a CNN-based predictor, which accepts the DFG as input and captures patterns that have high impact on area in order to predict the area savings. Recent years have seen great success for CNNs in the area of image recognition, owing to their ability to identify and detect spatial features that distinguish one object from another. A CNN's ability to learn important spatial features without specifically being told what those features are makes this approach desirable, since it is portable across different downstream CAD tools and target devices by retraining the CNN. Sections V and VI delve into details of the CNN-based predictor.

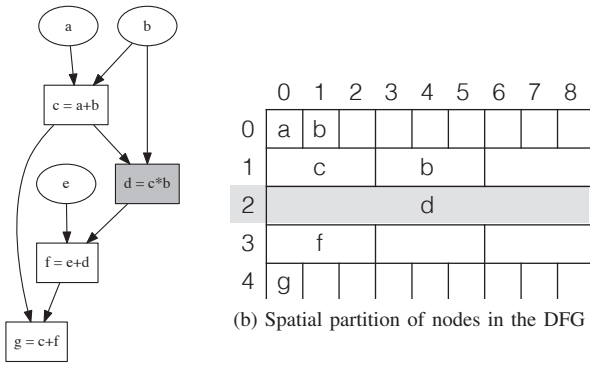
V. Spatial DFG Representation

In order to build a CNN-based predictor, a key problem we have to overcome is how to go about presenting a DFG to a CNN as input. Inspired by the success of CNNs in image recognition, we represent the DFG connectivity and metadata *spatially* as an image with 3 channels, elaborated upon below.

Recall that variables correspond to nodes in the DFG and data dependences correspond to edges. To estimate the area impact of reducing a specific variable's bitwidth, we take a *window* of the DFG surrounding that variable, and represent the DFG window (a portion of the entire DFG) as an image. Inputs to a CNN must generally be of fixed size, and thus, the size of our window is also fixed to include a portion of the DFG in the transitive fanout and transitive fanin of the bitwidth-reduced variable. We refer to the variable being considered for bitwidth reduction as the *target variable*. The natural question that arises is: how big should the window (and associated spatial image representation) be? If the target variable's bitwidth were reduced, those reductions may propagate to reductions of variables in its transitive fanin and fanout, particularly to nearby nodes of the DFG in the local vicinity of the target variable. It is desirable if the window size is large enough to "capture" such bitwidth propagations.

To determine the window size, we study the typical *connectivity* and *depth* of nodes in a DFG. We first note that in the LLVM IR, most instructions are either unary or binary, with the exception being `call`, `select` and `phi` instructions having 3 or more operands. Thus, most nodes in the DFG have fanin ≤ 3. Regarding typical node fanout, we profile a suite of benchmarks (CHStone [6]) to obtain the fanout distribution of IR nodes, shown in Fig. 3. Observe that 95% of all IR variables have 3 or fewer fan-outs. Given these observations, we opt to support a connectivity of 3 for both the transitive fanins ($C_{in} = 3$) and the transitive fanouts ($C_{out} = 3$) of the target variable, capturing up to three predecessor nodes of each node in the transitive fanins of the target variable and up to three successor nodes of each node in the transitive fanouts of the target variable in our spatial DFG representation.

We also study the typical depths of DFGs, again using the CHStone benchmark suite. We profile the maximum number of DFG nodes needed to be traversed before reaching a target variable, as well as the maximum number of nodes needed to be traversed from the target variable to reach a leaf node of the DFG. Figs. 4 and 5 show the distribution of these numbers, respectively. These two figures represent the upper bound on the number of DFG levels to examine above and below a target variable in order to capture all changes in the bitwidth for the CHStone benchmarks. In our experimental study, we choose a depth of 5 for both the transitive fanins ($D_{in} = 5$) and transitive fanouts ($D_{out} = 5$), empirically determined to produce reasonable results. Given fixed parameters of fanin connectivity (C_{in}), fanin depth (D_{in}), fanout connectivity (C_{out}), and fanout depth (D_{out}), the maximum size of the DFG window is $(\sum_{i=1}^{D_{in}} {C_{in}}^i) + (\sum_{i=1}^{D_{out}} {C_{out}}^i) + 1$ nodes,



(a) DFG

(b) Spatial partition of nodes in the DFG

Fig. 6: An example of mapping a DFG to an image with $C_{in} = C_{out} = 3$ and $D_{in} = D_{out} = 2$.

where the 1 is the target variable itself.

We now introduce our spatial DFG representation that maps a DFG window to an image, which is best explained visually using an example. Fig. 6 shows a DFG and its corresponding spatial representation (image) with $C_{in} = C_{out} = 3$ and $D_{in} = D_{out} = 2$. In this case, node d is the target variable (shown shaded). All pixels in the middle row of the image are assigned to node d. Rows above the target variable correspond to DFG nodes in the transitive fanin of the target variable; rows below the target variable correspond to DFG nodes in the transitive fanout of the target variable. On the fanin side, pixels directly above a node in the row above are partitioned into C_{in} sections, holding up to C_{in} operands of the node. d has two operands, c and b, appearing above d in the image, each occupying three pixels. c has two operands, a and b, represented by one pixel each in the top row. The bottom half of the image represents d's transitive fanout in a similar way.

Given the above spatial mapping of the DFG nodes to pixels in the image, we choose three pieces of information about each DFG node to encode into the three channels of the image: 1) the original bitwidth of each variable, 2) the number of bits reduced as a result of reducing the target variable's width, and 3) the type of operation (e.g. `add`, `mul`, etc.) represented numerically as an opcode. Regarding channel 3, the numerical opcode we use for each operation type corresponds to the operation's typical area consumption (i.e. we use higher opcodes for large-area operations such as division). As we will show in our experimental study, this spatial representation contains enough information for the CNN to learn/extract features and distinguish between bitwidth reductions that are favourable vs. unfavourable to area.

VI. Experimental Study

A. Methodology

To evaluate the effectiveness of Sensei, we train the CNN-based predictor with data points from the CHStone benchmark suite [6], comprising applications from a variety of domains, including media processing, security and microprocessor simulation. Ten out of twelve of the CHStone benchmarks are included in this study; `adpcm` and `jpeg` are omitted since they require more DSP blocks than available on the Altera Cyclone V device we target. For every benchmark in CHStone, we enumerate all IR variables that correspond to a `C` program variable. For each such IR variable, we generate variants of the benchmark by reducing the width of the IR variable to each of these values: 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, and 64 bits, up to the variable's original width.

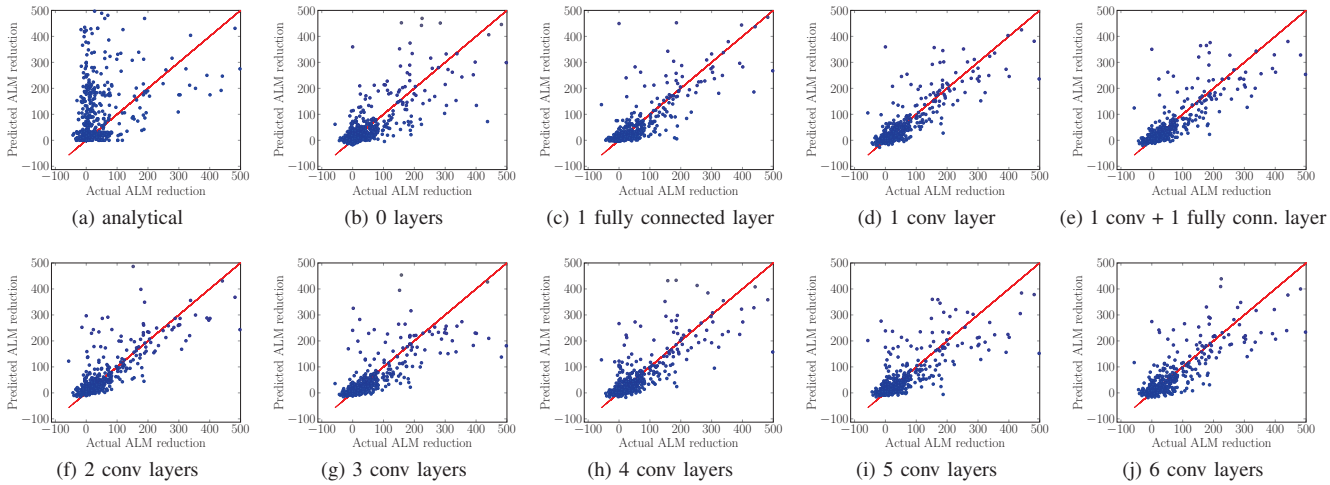
With this method, we generate ∼7000 bitwidth-reduced circuits to be used as CNN training and test cases (i.e. separate benchmark circuits wherein one IR variable's bitwidth is reduced by a given amount), each of which are generated by LegUp HLS and pushed through Quartus II to obtain the detailed resource usage report. In this work, we focus on estimating adaptive logic module (ALM) area consumption in the Altera Cyclone V 28nm FPGA, where an ALM is a dual-output 6-LUT with two flip-flops. Future work will consider the area of other block types, e.g. DSP units and block RAMs.

To evaluate our CNN predictor, we partition the entire data set into 10% test set and 90% training set. For each benchmark, we randomly assign 10% of the data to the test set, and the rest are assigned to the training set. This yields 6400 data points in the training set and 640 data points in the test set. We use Caffe [7] to train the network for 1000 epochs with different network architectures, discussed in the next sub-section.

B. CNN Model Design

We explore different CNN models, as there is no known CNN model that has been proven to work well on this type of pseudo-image. All CNN architectures we explore have one fully connected layer at the end of the network, generating a real-valued output prediction of the number of ALM reduction. We vary the number of convolutional layers and additional fully connected layers before the default fully connected layer to evaluate their impact on prediction accuracy. Inspired by the success of the VGG-16 CNN in image recognition [14], which uses 3×3 convolutional filters, we also use filters of this dimension in this work. As we consider CNNs having similar models to those in image recognition, we leverage prior work in the community on optimizing the performance of such CNNs. Fig. 7 shows the prediction accuracy on the test set as we modify the type and number of layers between the input and the default fully connected layer at the end. The y-axis shows the predicted number of ALMs reduced, and the x-axis shows the actual number of ALMs reduced after Quartus II synthesis, place and route. The ideal case is to have all points line up at the red line, which corresponds to the $y = x$ line. The analytical approach in Fig. 7a is discussed later in Section VI-C.

Looking at the plots in Fig. 7, if we compare the models with no convolutional layers (Fig. 7b and Fig. 7c) and one convolutional layer (Fig. 7d), we observe an improvement in the quality of results (QoR). However, as the number of convolutional layers increases (Fig. 7f-7j), the network overfits the training data, and we see a QoR degradation. This is likely attributed to the relatively small amount of



(a) analytical (b) 0 layers (c) 1 fully connected layer (d) 1 conv layer (e) 1 conv + 1 fully conn. layer

(f) 2 conv layers (g) 3 conv layers (h) 4 conv layers (i) 5 conv layers (j) 6 conv layers

Fig. 7: CNN architecture exploration results with the number of ALMs reduced. Each graph shows the predicted ALM reductions vs. the actual ALM reductions. (a) shows the analytical approach, while (b) to (j) shows the results as we vary the layers between the input layer and the final fully connected layer.

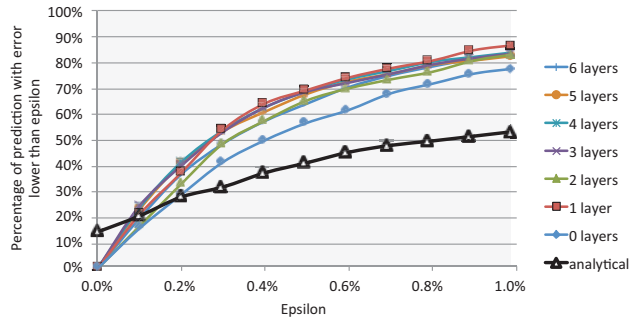


Fig. 8: QoR as we vary the error threshold ϵ. Each line represent a network with different number of convolutional layers between the input and the final fully connected layer.

training data that we have in this study. We expect that as the number of datapoints used in training increases (which would be commonplace in an industrial context where many more benchmark designs are available), the network would require more convolutional layers before overfitting.

C. Prediction Accuracy Results

To validate that Sensei's prediction results are reasonable, we also implemented a simple *analytical* predictor based on a typical approach for estimating area during HLS as follows: We first characterized the area savings per bit reduced for each type of operation (e.g. add, subtract, shift, logical) in isolation, using Verilog microbenchmarks. Then, during prediction, the projected area savings owing to bitwidth reductions is obtained by 1) analyzing the number of bits reduced for each operation type, 2) consulting the characterization data, and 3) summing the area savings of all operation types.

To quantify the prediction accuracy, we compute the error per datapoint as the absolute difference between the predicted and the actual ALM reductions, normalized to the number of ALMs in the overall circuit[2]. We define a threshold ϵ (percentage of the overall circuit area) that represents the maximum error for an acceptable prediction. We then compute QoR as the percentage of all datapoints having error lower than ϵ. Fig. 8 shows the QoR of the analytical and the CNN-based predictors when we sweep ϵ from 0 to 1% of the original circuit area. For the analytical predictor, we see that 53% of the datapoints have an error lower than 1%, compared to the 87% from our best CNN-based predictor (1 convolutional layer). Fig. 7a shows the predicted vs. actual ALM reduction on the same test set for the analytical predictor. Unlike our CNN-based predictor, many of the points with the analytical predictor lie far above and below the ideal line. The majority of these mispredictions are due to over-estimating the ALM reduction, since the formulation of the analytical predictor aggregates area reductions for each type of operation separately. It is unaware, for example, that connected operations in the DFG may ultimately be covered by a single look-up table in the FPGA mapping. It is also unaware that by reducing the widths of certain variables, operations that used to be bound together may no longer be able to share a functional unit, incurring extra area. The data-driven learning method used in the CNN-based predictor is able to overcome these limitations, given the same amount of information as the analytical predictor.

D. Ranking Prediction Results

To demonstrate the efficacy of Sensei in ranking the top variables, Fig. 9 shows the potential for area savings using the top results from the predictions of our test set. For the CNN-based predictor, we use the architecture with one convolutional layer since it yields the highest QoR at $\epsilon = 1\%$. Our metric for the area savings potential is the number of ALMs reduced

[2]With the CHStone benchmarks used in this study, the total number of ALMs ranges from 1200 to 9000.

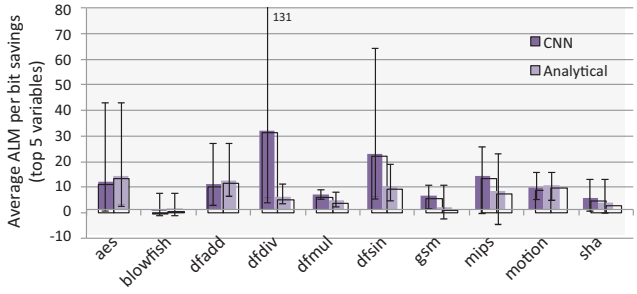


Fig. 9: ALM per bit savings averaged across predicted top 5 variables. Each bar shows the average, and each accompanying line shows the lowest and highest savings of the 5 variables.

per bit reduced. We compute this by taking the top 5 variables selected by the CNN-based predictor and measure their actual area savings (averaged across the 5 variables). We also show the lowest and highest savings among the predicted top 5 variables. We compare this against the top 5 variables selected by the analytical predictor. For 6 of the 10 benchmarks, following the *advice* of the CNN-based predictor yields better area savings (up to 6× for `dfdiv`). For the remaining 4 circuits, the difference between the two predictors is minor. On average, the CNN-based predictor reports variables that yield higher area savings than the analytical predictor.

Overall, the results are quite encouraging, as we believe this to be the first work to apply CNNs to predict post-synthesis area in HLS. The CNN-based predictor achieves superior accuracy and ranking ability, compared with the analytical approach, which is a typical way of assessing area in HLS.

VII. Related Work

In this work, we explore fine-tuning the software description of circuits that are synthesized via HLS. This is akin to DSE in HLS where frameworks explore different ways to write high-level code to improve results. The need for efficient DSE for HLS has been tackled from multiple angles in literature. So et al. [15] focus on loop nest computations and present a compiler-based DSE algorithm that applies loop transformations to obtain different variants of the design; the best design is selected by using synthesis estimation techniques that balance the computation intensity with the memory access rate. Kulkarni et al. [8] iteratively apply aggressive compiler tranformations on the data flow graph of a program and use compile-time estimations to evaluate the area impact of varying design alternatives.

Many commercial HLS tools provide support for directives in the form of pragmas as a means for the user to fine-tune specific parts of their design. Carrion Schafer and Wakabayashi [12] use GA-ML, a genetic algorithm based on a predictive model generated from machine learning techniques, to find the Pareto-optimal set of directives for a given design. Liu and Carloni [10] employ the Random forest learning model to iteratively refine the Pareto set of HLS directives. Instead of randomly sampling the design space, they use Transductive Experimental Design to select a representative set of designs to make up the training set. Lo and Chow [11] consider the problem of automatically selecting an optimal set of HLS directives using sequential model-based optimization (SMBO) methods. To the authors' knowledge, this work is the first to apply CNNs in HLS for prediction.

VIII. Conclusions and Future Work

In this paper, we present a novel take on area prediction in HLS, employing a CNN to predict post-synthesis area reduction using the DFG of a program. We evaluate this idea using the LegUp HLS tool and Caffe, exploring a variety of CNN architectures. Using this accurate area-prediction scheme, we develop Sensei, an area-reduction advisor that brings the HLS user's attention to program variables that result in the most post-synthesis area savings. This addresses the inefficiency that arises in HLS, when the high-level language imposes standard overprovisioned datatypes even though the underlying hardware can support arbitrary-width datapaths.

For future work, a natural extension is to extend the predictor to consider DSP blocks and perhaps RAM usage as well. Beyond this, we believe there to be tremendous scope for application of CNNs in CAD, particularly HLS. It may be possible, for example, to train CNNs to accurately estimate post-routing circuit delay or power consumption at the HLS stage, thereby enabling better decisions to be made both in the tools themselves and by human engineers.

References

[1] U. Aydonat *et al.*, "An opencl deep learning accelerator on arria 10," in *ACM FPGA*, 2017.
[2] A. Canis *et al.*, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 2, 2013.
[3] P. Coussy *et al.*, "An introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 26, no. 4, 2009.
[4] M. Gort, "Fast cad for fpgas," Ph.D. dissertation, University of Toronto, June 2014.
[5] M. Gort and J. H. Anderson, "Range and bitmask analysis for hardware optimization in high-level synthesis," in *IEEE/ACM ASP-DAC*, 2013.
[6] Y. Hara *et al.*, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis," *Journal of Information Processing*, vol. 17, 2009.
[7] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
[8] D. Kulkarni *et al.*, "Fast area estimation to support compiler optimizations in fpga-based reconfigurable systems," in *IEEE FCCM*, 2002.
[9] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *IEEE/ACM CGO*, 2004.
[10] H. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *ACM/IEEE DAC*, 2013.
[11] C. Lo and P. Chow, "Model-based optimization of high level synthesis directives," in *FPL*, 2016.
[12] B. C. Schäfer and K. Wakabayashi, "Machine learning predictive modelling high-level synthesis design space exploration," *IET Computers & Digital Techniques*, vol. 6, no. 3, 2012.
[13] J. Schmidhuber, "Deep learning in neural networks: An overview," *CoRR*, vol. abs/1404.7828, 2014. [Online]. Available: http://arxiv.org/abs/1404.7828
[14] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556
[15] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *ACM PLDI*, 2002.
[16] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *ACM FPGA*, 2017.