REHLS: Resource-aware Program Transformation Workflow for High-level Synthesis

Atieh Lotfi
UC San Diego, CA, USA
alotfi@cs.ucsd.edu

Rajesh K. Gupta
UC San Diego, CA, USA
gupta@cs.ucsd.edu

***Abstract*—Despite considerable improvements in existing HLS tools, they still require designer interventions to provide efficient synthesis results. This manual design space exploration and code rewriting and optimization takes significant time and negates the HLS design productivity gains. To overcome this challenge, this paper uses compiler frontend as an independent preprocessing step to explore the design space and adds an automated source-to-source transformation step before HLS. In particular, it shows how inherent regularity in applications can be used to construct a workflow that analyzes the program, explores the design space for resource optimization opportunity, and transforms the program accordingly. When the transformed program is synthesized using the HLS tool, it uses less hardware resources with similar latency comparing to the original design. The synthesis results on a modern Xilinx Virtex-7 FPGA for a diverse set of applications show that our automated transformation can reduce the design area by an average of 15.4% with less than 1% performance overhead compared to the state-of-the-art Xilinx HLS tool solutions. This automated tool reduces the design time and especially can be useful for non-expert FPGA designers.**

I. INTRODUCTION

High-level synthesis (HLS) tools help improve productivity by raising the programming abstraction from hardware description languages to higher level languages like C, C++, OpenCL, or SystemC. Despite the advances in HLS design automation [1], the quality of synthesis results is still not comparable to hand-coded RTL designs, requiring an expert designer effort to optimize and rewrite the source code especially when a design pushes resource limits. Designers often need to manually explore a large design space to find the best design option among many different design alternatives. This manual exploration and code transformations takes significant time, and requires knowledge of hardware microarchitecture and the coding style of HLS tool, which negates the HLS design productivity gains.

This paper presents a practical source-level design exploration and mapping workflow that enables automatic source-to-source transformation techniques to improve the efficiency of synthesis result in an HLS design flow. Different system-level optimizations and code restructuring can be applied on a given application specification, where each transformation impacts differently on resource utilization and performance of the design after synthesis. In this paper, we focus on a class of code transformations that results in improving hardware resource utilization without noticeable overhead on design latency. These transformations seek to reuse the same instance of a hardware component for parts of the design that have recurring sequence of operations, or computational *patterns* [2]. Indeed, an intelligent use of common computational patterns or *regularities* is a key reason why the manual design and optimization often excels the design synthesized by automated HLS tools.

Expanding the automated design space exploration to detect regularities at a higher level design can reduce the design time and improve the design quality. At the same time, excessive resource sharing decisions could introduce more overhead than benefit due to need for time-multiplexed control that can be expensive on FPGA implementation targets.We present an automated pre-synthesis regularity extraction workflow, called REHLS, that identifies the program inherent regularities that are not automatically detected by HLS tools, and evaluates the effectiveness of sharing resources for instances of those patterns. This information is used to automatically modify the source code in a way that guides the HLS tool to perform resource sharing for the selected parts of the code. Using this modified code, the HLS tool can provide a more efficient solution that consumes less hardware resources (with similar latency) when synthesized on FPGA. This reduces design time and especially helps non-expert designers create a more efficient design.

The rest of this paper is organized as follows. Section II surveys prior work in this topic area. Our resource-aware program transformation workflow is presented in Section III. In Section IV, we present experimental results, followed by conclusion in Section V.

II. RELATED WORK

In recent years, some source-to-source transformation tools have been developed to perform program optimizations that are not automatically done by the HLS tools. Some tools target polyhedral loop transformation [3] and memory partitioning [4]. Others target automated expression simplification and bitwidth optimization [5][6]. In contrast to these works, our tool targets detecting computational patterns that can benefit from resource sharing.

Regularity extraction has been studied extensively in application-specific instruction set processors [7] and synthesis literature [8]. Some works extract regularities from behavioral specification in HLS context. In these works, after discovering patterns, the scheduling and resource binding algorithms of synthesis flow are changed to reduce the resource usage of the generated design[8][2]. Despite these proposals, off-the-shelf HLS tools still can not automatically exploit *non-obvious* regularities in the design; and it is not possible to change synthesis flow and algorithms in commercial HLS tools. Therefore, we apply the required

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.92

IEEE computer society

modifications on the high-level source code itself to guide the HLS tool to efficiently share hardware resources when useful.

III. REHLS WORKFLOW

Given a C program with inherent computational patterns, REHLS explores the opportunities for design optimization through resource sharing, and automatically transforms the program so that the HLS tool generates a more efficient design after synthesis. Fig. 1 illustrates an overview of the proposed workflow. First, the program is converted to LLVM intermediate representation (IR) [9]; the analysis and transformation is done through LLVM passes; and finally the transformed LLVM IR is converted back to a C program. The transformed C program, along with a generated directive file, guides the HLS tool to share hardware resources when useful.

A. Program Analysis

Given an input C program, the first step is to detect patterns in the program that are candidates for resource sharing, and further select some of them for sharing that result in reducing resource usage with no negative effect on latency of the synthesized design.

1) Pattern Detection

REHLS finds all patterns that are repeated across different basic blocks in each function in the program. To do that, the data flow graph (DFG) of different basic blocks are extracted from the program's LLVM IR. DFG is a directed acyclic graph G(V, E), in which nodes in V represent operations, and edges in E represent data dependencies between operations. Our goal is to find common subgraphs among these DFGs. (The common subgraphs are called pattern.) Given a pair of DFGs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the goal is to enumerate all the subgraphs of G_1 that are isomorphic to subgraphs of G_2, where functionally, type, and bitwidth of corresponding nodes of V_1 and V_2 are the same.

Our algorithm detects patterns with a breadth-first search approach. Our subgraph enumeration process is incremental, meaning that size *i+1* subgraphs are enumerated when all the size *i* subgraphs are enumerated. If a size *i* subgraph is not frequent enough, it is removed and no further considered for creating new subgraphs of size *i+1*. In each iteration *i*, our algorithm finds all patterns of size *i* (with *i* nodes) across different DFGs. To do so, in iteration *i*, we extend the detected patterns of size *i-1* by adding one of their neighbor nodes in DFG. This method ensures that we only investigate those subgraphs that can be a potential pattern. For each subgraph of size *i*, if the number of occurrences across different DFGs are more than an acceptable *frequency limit*, we add it to the set of candidate patterns. Otherwise, it is not considered for finding larger subgraphs. This process is repeated until either no more new pattern is found or we enumerate the subgraph with maximum possible size.

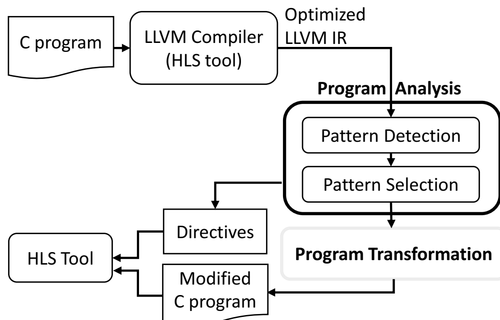

Fig. 1: Overview of REHLS, our resource-aware regularity extraction workflow

2) Pattern Selection

Each detected pattern is a candidate component for resource sharing. The goal is to select all patterns that sharing hardware resources for their instances can reduce the design area with negligible change in latency comparing to the baseline design. To achieve this goal, we estimate the effectiveness of resource sharing for each detected pattern.

If hardware resources are shared for instances of a pattern, the corresponding parts of the program, that have instances of that pattern, must be run sequentially. Therefore, if instances of a pattern have overlapping lifetimes in the baseline design, the design latency increases after resource sharing. Therefore, to meet the performance requirement, we only select instances of a pattern with no overlapping lifetime. For each detected pattern, we use an LLVM dependency analysis pass to keep only those instances that have dependency and, therefore, can not be executed at the same time. This way, we ensure that resource sharing will not noticeably increase the design latency. It should be noted that adding multiplexers might make the critical path longer, but this change is usually not considerable.

In this paper, we use a greedy algorithm for pattern selection. At each step, the best pattern is chosen based on an area gain metric. For each pattern, we estimate the effect of resource sharing on the resource utilization of design. In general, sharing FPGA resources can save area. However, multiplexers are introduced in the inputs of the shared components. These multiplexers (especially larger ones) have non-negligible area. Therefore, the granularity, frequency, and type of shared component determines if sharing is beneficial in terms of reducing area or not. Because multiplexers are only added in the inputs of shared components, more complex and more frequent patterns are better candidates for resource sharing. When any candidate pattern P_i is selected for sharing, instead of F_{Pi} instances of the allocated hardware unit, the new design has only one instance of that hardware unit plus some extra multiplexers. For each candidate pattern P_i, we compute an estimation of the area gain that can be achieved due to resource sharing for the selected instances of that pattern:

$$AreaGain_{Pi} = (F_{Pi} - 1) \times Area_{Pi} - N_{inputsPi} \times Area_{mux} \quad (1)$$

F_{Pi} is the frequency (number of instances) of the candidate pattern P_i that has area $Area_{Pi}$, and $N_{inputsPi}$ is the number of inputs of the candidate pattern. $Area_{mux}$ is the area of a multiplexer of required bitwidth. From the equation, the more complex and larger pattern with more frequency has larger area gain. In this equation, area is defined as the number of hardware elements (FF, LUT, DSP48) that the pattern component uses. In this paper, we only share the logic

elements, therefore, the number of BRAM elements does not change. Using equation 1, we estimate the number of reduced (or increased) FF, LUT, and DSP48 elements due to resource sharing. We calculate the *areaGain* for all patterns, and remove those that have negative gain. For the remaining set of patterns that have positive *areaGain*, in each iteration, a pattern with largest *areaGain* is selected.

After selecting one pattern, we still continue to select other patterns that are good for resource sharing. First, we remove all subgraphs that have overlapping node with any node in the instances of the selected pattern. Then we continue our search for the remaining subgraphs in the pattern set. We continue this process until no more candidate pattern (with positive *areaGain*) is left in the pattern set. At the end of this phase, we find all independent patterns and their instances that are useful for resource sharing.

B. Program Transformation

After selecting patterns, the next step is to make the required changes in the code. The transformation is done through an LLVM pass that modifies the LLVM IR. We define operations inside the pattern as individual functions, and call this function instead of calling the operator in *C* for each instance of the pattern. To guide the HLS tool to share resources for instances of the pattern, the *allocation* pragma is used. ("#pragma HLS allocation instances=patternOP limit=1 function"). These pragmas are written in a directive file which is given to the HLS tool as input. The transformed LLVM IR is then converted back to C using a resurrected LLVM C Backend. The generated code is a low-level C code that can be synthesized using HLS tool.

IV. Experimental Results

A. Implementation and Experimental Setup

We use Xilinx Vivado HLS Suite 2015.4 [10] as an exemplary state-of-the-art tool for our experiments. This tool uses the LLVM compiler and compiles a behavioral C/C++ program into RTL hardware design. Before synthesizing the program, it performs special LLVM passes to optimize the IR. Although we targeted Vivado HLS for our experiments, the proposed workflow is applicable to any LLVM-based HLS tool.

We evaluate REHLS using a set of computation kernels and applications. A description of each benchmark can be found in Table I. For each benchmark, the number of selected patterns (*NP*), the size of selected patterns (or the number of operations in the pattern) (*PS*), and the frequency of each pattern (*PF*) are reported in the table. Xilinx Virtex-7 XC7V585T FPGA device is used as the target hardware platform in our experiments.

B. Results

Table II demonstrates our experimental results. For each benchmark, we report the resource utilization, performance, and the number of replicas of the design that can be fitted on FPGA for three different *implementations* of the program. For each benchmark, the top two rows reflect synthesis results for the baseline and REHLS-optimized programs (*Baseline* and REHLS rows in the table). We also tried to manually

TABLE I: Benchmark Descriptions

Benchmark	Description	NP	PS	PF
adi	Alternating direction implicit solver	2	16,2	2,2
idct	Inverse discrete cosine transform	3	13,13,16	2,2,2
gauss	3D gaussian convolution	1	16	4
bnn	4-layer bitwise neural net(feedforward)	2	6,5	4,3
jacobi	Jacobi iterative method	1	8	4
sort	Radix sort	1	4	4

optimize each program to reduce the resource utilization, the related results are shown in the *Hand-opt* row. The fourth row of each benchmark (*Improvement*) shows the relative improvement of the REHLS optimized version over the baseline program. (Improvement numbers for resource utilization and performance is the percentage of relative reduction, and for the number of replicas is the percentage of relative increase.) For these benchmarks, the selected patterns are usually from inside the loop bodies. In our experiments, loops are pipelined.

1) Area Savings

Columns *LUT*, *FF*, and *DSP* in Table II show the resource utilization for the baseline, REHLS-optimized, and hand-optimized programs. Our transformation does not change the number of memory elements, therefore, the number of utilized BRAM elements is not reported. Our results indicate that REHLS reduces the number of utilized resources on an average of 15.4% (9.9% LUT, 17.9% FF, and 20.6% DSP elements) comparing to the baseline design. This is achieved by the detection of patterns and reusing the same hardware resources for them. The amount of reduction depends on different factors, including the type and number of operations in the pattern, and the percentage of resource utilization of instances of patterns comparing to the other parts in the baseline program. For example, in *adi*, the selected patterns form the major computations in the design, and therefore, sharing hardware resources reduces its area by 44%. On the other hand, the selected patterns in *bnn* are not that much complex comparing to other computations in the design, therefore, it can only achieve 1.8% area reduction.

We also compared the resource utilization for REHLS-optimized and hand-optimized programs. In most benchmarks, the synthesis result for REHLS-optimized program is similar to the hand-optimized results. In case of *bnn*, REHLS can not find the large pattern that we found by hand. The manually-found pattern includes conditionals and *for* loops, which is beyond the scope of DFGs. In some other benchmarks (e.g. *gauss*), we could reuse some of the hardware cores for other operations in the design that are not part of a pattern. This results in a small improvement in resource usage of hand-optimized program comparing to the REHLS-optimized.

2) Performance

Column *Performance* in Table II shows the execution time (*ns*) (clock period $\times$latency) for the baseline, REHLS-optimized, and hand-optimized designs after synthesis. As can be seen, performance changes of REHLS comparing to baseline design are negligible (below 1%), because we only share resources for those instances of patterns that have dependency. The slight increase in performance comes from

TABLE II: Experimental results on Virtex-7 FPGA

BM	Implementation	LUT	FF	DSP	Performance	Replica
adi	Baseline	34440	34012	176	1969760302	7
	REHLS	19240	17792	104	1987776833	12
	Hand-opt	18824	17556	104	1987776833	12
	Improvement	44.13%	47.68%	40.9%	-0.91%	69%
idct	Baseline	2421	2118	64	3246	19
	REHLS	2257	1344	32	3277	39
	Hand-opt	2129	1283	28	3298	45
	Improvement	6.77%	36.5%	50%	-0.9%	105%
gauss	Baseline	15312	10145	661	10791672	1
	REHLS	15274	8380	565	10792250	2
	Hand-opt	15080	8361	501	10792481	2
	Improvement	0.24%	17.4%	14.5%	-0.005%	16.9%
bnn	Baseline	1557	679	0	98274361	233
	REHLS	1489	670	0	98327691	244
	Hand-opt	921	376	1	86416466	395
	Improvement	4.36%	1.32%	0%	-0.05%	4.5%
jacobi	Baseline	18331	7241	174	21769522422	7
	REHLS	18075	7113	142	21792855244	8
	Hand-opt	18361	7079	110	21792855244	11
	Improvement	1.39%	1.76%	18.39%	-0.1%	14.28%
sort	Baseline	1297	481	0	1954053.05	280
	REHLS	1263	466	0	1964197.6	288
	Hand-opt	1263	466	0	1964197.6	288
	Improvement	2.6%	3.1%	0%	-0.5%	2.8%

the delay that multiplexers might cause.

3) Throughput Speedup

Traditional FPGA design flows usually follow a two-step approach. First, a given application is optimized for best performance and resource utilization. Then the optimized design can be replicated and executed in parallel to fully utilize the available capacity of the target FPGA, and to improve throughput [11]. As shown in Section IV-B1, REHLS reduces the area of synthesized design for our benchmarks. Therefore, the number of parallel replicas of that design, that can be fitted in the fixed area budget of the FPGA is increased. In addition, because the effect of our transformations on latency is negligible, the increase in the number of mapped applications on FPGA, results in higher throughput. The number of replicas that can be fitted in our target FPGA for each design version is shown in the *Replica* column of Table II. For each benchmark, we can increase the number of design replicas on FPGA until one of the resources available on FPGA reaches its maximum limit. Considering the geometric mean across all the benchmarks, REHLS improves the number of mapped kernels by a factor of 1.43× (maximum 2× in *idct*). For the hand-optimized version, we can increase the number of replicas by 19% on average when comparing to the REHLS-optimized program.

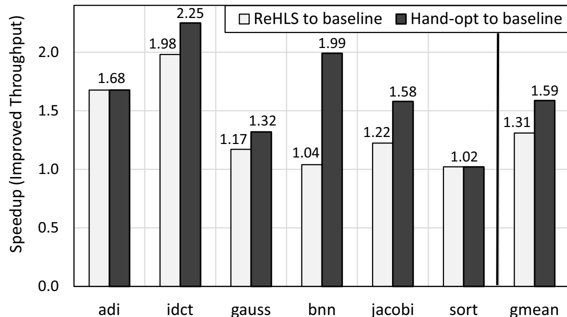

Fig. 2: Throughput speedup (Normalized to baseline)

Fig. 2 shows the corresponding throughput speedup results – throughput of the REHLS and hand-optimized designs normalized to the throughput of the baseline design. As shown, REHLS achieves on average 1.31× higher throughput (between 1.02× and 1.98×) comparing to the baseline design. For the hand-optimized program, we can achieve 21% (59%) throughput speedup comparing to the throughput of the REHLS-optimized (baseline) program.

It should be noted that generating the hand-optimized version of programs require extensive exploration and synthesis of different program versions to find which operations are good targets for sharing. Our automated tool can significantly reduce the time and effort required for optimization.

V. Conclusion and Future Work

Preparing the code for HLS tools with high-level transformations is not new, but it is mainly done manually. However, some of these transformations can be done efficiently, faster, and error free in an automatic way by a front-end compiler. In this paper, we presented an automated resource-aware regularity extraction workflow. This pre-HLS workflow reduces the resource usage by identifying and exploiting inherent computational patterns in an input program. Experimental results over a variety of benchmarks show that our method is able to achieve 15.4% reduction in resource utilization and consequently 1.3× throughput speedup over a best-in-class commercial HLS tool targeting Xilinx FPGAs. In the future work, we focus on finding patterns with larger granularity as well as patterns that are functionally similar.

VI. Acknowledgments

This work was supported by the DARPA craft program under award HR0011-16-C-0037.

References

[1] J. Cong et al, “High-level synthesis for fpgas: From prototyping to deployment,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[2] S. Hadjis et al, “Impact of fpga architecture on resource sharing in high-level synthesis,” ser. FPGA ’12, 2012.

[3] L.-N. Pouchet et al, “Polyhedral-based data reuse optimization for configurable computing,” ser. FPGA ’13, 2013.

[4] Y. Wang et al, “Memory partitioning for multidimensional arrays in high-level synthesis,” ser. DAC ’13. ACM, 2013.

[5] X. Gao et al, “Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis,” ser. FPGA ’16. ACM, 2016.

[6] A. Lotfi et al, “Grater: An approximation workflow for exploiting data-level parallelism in fpga acceleration,” ser. DATE ’16, 2016.

[7] P. Brisk et al, “Instruction generation and regularity extraction for reconfigurable processors,” in *CASES ’02*, 2002.

[8] J. Cong, H. Huang, and W. Jiang, “A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis,” in *DATE*, 2010.

[9] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *CGO ’04*, 2004.

[10] “Vivado high-level synthesis,” http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[11] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, “Resource-aware throughput optimization for high-level synthesis,” ser. FPGA ’15, 2015.