ScaleHLS: Achieving Scalable High-Level Synthesis through MLIR

Hanchen Ye$^1$, Cong Hao$^2$, Hyunmin Jeong$^1$, Jack Huang$^1$, Deming Chen$^1$

$^1$University of Illinois at Urbana-Champaign, $^2$Georgia Institute of Technology

ABSTRACT

High-level Synthesis (HLS) has been widely adopted as it significantly improves the hardware design productivity and enables quick design space exploration (DSE). However, existing HLS tools do not scale well to large designs for two main reasons: (1) The intermediate representations (IR) are not initially designed for HLS, thus are not expressive enough for comprehensive HLS design spaces; (2) The traditional HLS algorithms are based on a single-level abstraction, thus cannot easily capture the design hierarchy and are not scalable as the design size grows. To tackle these problems, we present ScaleHLS, a new HLS compilation flow based on a multi-level compiler infrastructure, MLIR. Utilizing MLIR, ScaleHLS introduces a hierarchical representation mechanism for HLS designs, enables scalable optimizations at multi-level abstractions, and directly generates optimized synthesizable HLS code. This approach not only explores the hierarchical design space efficiently but also scales well to large HLS designs. The initial experiments show that comparing to the baseline designs only optimized by the regular LLVM optimizations of Xilinx Vivado HLS, ScaleHLS improves the performance by up to 768.2× on computation kernel level algorithms and 4107.6× on a neural network model MobileNet-v2.

1 INTRODUCTION

High-level synthesis (HLS) technique automatically translates high-level languages to dedicated hardware accelerators, thereby eliminating the cumbersome and error-prone programming of hardware description languages [11]. Recent years, HLS has been widely used in many applications, including neural networks [1], IoT [13], video processing [7], etc. These HLS designs highly rely on user-specified directives and manual code transformation to improve the quality of hardware. However, as HLS tools open up large design spaces, non-ideal design choices may easily lead to sub-optimal solutions and poor overall performance.

Recently, we have witnessed a large number of papers investigating the automatic quality of results (QoR) estimation and design space exploration (DSE) for HLS. Authors of [14–16] extracted necessary design information from static dataflow graphs or dynamic execution traces, then passed such information to predefined analytical models for generating the estimation. Authors of [4, 8, 9] introduced machine learning methods to extract unique features that cannot be easily parameterized by analytical models. On top of QoR estimation, authors of [14, 15] proposed automatic DSE tools based on the guidance of resource and performance estimations. In addition, polyhedral analysis and integer linear programming (ILP)-based algorithms were exploited in [5, 17, 18] for proposing qualified design candidates and searching for the optimal solution under hardware resource constraints.

However, existing DSE methods are difficult to handle large HLS designs containing a large number of sub-modules and sophisticated inter-dependencies. The reason is that existing efforts heavily rely on unified and flattened intermediate representations (IR), such as LLVM, for conducting analysis and optimizations. Such a low-level IR barely supports hierarchical hardware optimization techniques, such as task/module level resource-sharing, scheduling and parallelization, inter-loop analysis and transformation, and hardware IP integration. These design optimizations are located at different abstraction levels and are very difficult to be explored and applied on a flattened low-level IR, thereby obstructing current approaches to comprehensively explore and optimize the large designs through HLS. Furthermore, the flattened IR of a large design will lead to a large non-hierarchical design space, which is hard to be effectively searched through the existing DSE algorithms.

To address the difficulty of handling large HLS designs and make the automatic DSE more scalable and flexible, we introduce ScaleHLS, a next-generation HLS tool which can represent and optimize large designs at multiple abstraction levels. The main contributions of this paper are:

- To the best of our knowledge, ScaleHLS is the first MLIR-based end-to-end HLS compilation flow.
- We propose a hierarchical and scalable optimization methodology, which optimizes HLS designs at multiple abstraction levels, including graph, loop, and directive levels, to handle the increasing design space as the HLS design size grows.
- We propose an automated DSE engine to search for the Pareto frontier of the important latency-area trade-off space. A QoR estimator is also developed to evaluate design points discovered by the DSE engine rapidly.
- We design a synthesizable HLS C++ emitter for bridging the gap between the MLIR compilation framework and RTL generation back-ends.

2 SCALEHLS FRAMEWORK

ScaleHLS is built on top of MLIR [2, 6], which is a compilation framework incorporating multiple levels of functional and representation hierarchy. ScaleHLS compiles programs described in high-level programming frameworks (e.g., ONNX [3] and PyTorch [10]) or general-purpose languages (e.g., C/C++) to synthesizable HLS C++ code. Figure 1 shows the architecture of the ScaleHLS framework, where Dialect is an MLIR terminology referring to a set of customized operations, types, and attributes. In the ScaleHLS compilation pipeline, the input programs are first parsed into MLIR constructed with tensor-level operations (e.g., ONNX and Aten...
We evaluate the ScaleHLS DSE engine on 6 computation kernels with a problem size of 2048 and the results are shown in Table 1. Xilinx Vivado HLS 2019.1 is used to generate the RTL code and the target platform is Xilinx XC7Z020 FPGA, which has 220 DSP slices and 345.9 Mb memories on chip. Table 1 lists the optimal unroll factors and pipeline initial intervals (II) discovered by the DSE engine. Loop perfection, loop order permutation, variable loop bound elimination, and array partition are also automatically applied for improving the design quality. Among all 6 benchmarks, a speedup ranging from 83.3x to 768.2x is obtained compared to the baseline design which is only processed by the regular LLVM optimizations of Vivado HLS. As previous DSE methods [14, 15] only supports single-level abstraction, they are difficult to find reasonable design points when the problem sizes are large. The proposed multi-level representation enables ScaleHLS to find previously unachievable design points and explore a more comprehensive design space.

To evaluate the ScaleHLS framework when handling large and complicated HLS designs, we take a MobileNet-v2 [12] PyTorch model as the test case and Xilinx VU9P FPGA which has 6840 DSP slices and 345.9 Mb memories as the target platform. To quantify the speedup contributed by each of the three optimizations (directive, loop, and graph) and evaluate the proposed multi-level optimization methodology, we perform ablation studies and the results are shown in Figure 2. From the data of D, L\{n\}, and G\{n\} denote directive, loop, and graph level optimizations, respectively. Larger \(n\) indicates stronger optimization.

Figure 1: ScaleHLS framework.

Table 1: DSE results of computation kernels.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Prob. Size</th>
<th>Speedup</th>
<th>Unroll Factors</th>
<th>Target II</th>
</tr>
</thead>
<tbody>
<tr>
<td>BICG</td>
<td>2048</td>
<td>83.3x</td>
<td>[32,8]</td>
<td>43</td>
</tr>
<tr>
<td>GEMM</td>
<td>2048</td>
<td>768.2x</td>
<td>[16,1,16]</td>
<td>6</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>2048</td>
<td>199.2x</td>
<td>[4,32]</td>
<td>9</td>
</tr>
<tr>
<td>SYR2K</td>
<td>2048</td>
<td>423.8x</td>
<td>[8,8,8]</td>
<td>29</td>
</tr>
<tr>
<td>SYRK</td>
<td>2048</td>
<td>542.3x</td>
<td>[32,4,8]</td>
<td>34</td>
</tr>
<tr>
<td>TRMM</td>
<td>2048</td>
<td>614.6x</td>
<td>[4,4,64]</td>
<td>25</td>
</tr>
</tbody>
</table>

3 INITIATIVE EXPERIMENTAL RESULTS

Figure 2: Ablation study results of MobileNet-v2. \(D, L\{n\},\) and \(G\{n\}\) denote directive, loop, and graph level optimizations, respectively. Larger \(n\) indicates stronger optimization.
REFERENCES


