High-level Synthesis: Can it outperform hand-coded HDL?

Date 2020-06-25
Authors
Stephane Gauthier
Zubair Wadood
Abstract

High-Level Synthesis (HLS), with its higher level of abstraction and faster validation capabilities, has long been touted as a more efficient way of implementing FPGA and ASIC designs. There is a common perception, however, that hand-coded HDL designs can outperform HLS. With the right tooling, the opposite can be true. In this white paper, we leverage Silexica’s SLX FPGA tool to optimize an industrial image processing application used for feature detection and tracking. Analyzing the C-based application with SLX FPGA helps us find the critical parallelization blockers and memory bottlenecks. Using these insights, we are able to quickly identify and refactor the sections of code that will provide us with better hardware-aware code. Afterwards, SLX FPGA is used to further optimize the application by automatically exploring the design space and inserting HLS pragmas. The whole process, implementation, optimization, and validation took us one week. The results are compared to a handwritten HDL version which took two months to develop. Our final HLS implementation optimized with SLX FPGA is 64% faster compared to the handwritten HDL implementation.
1 Introduction

FPGA Design Challenges

As FPGA densities continue to grow with shrinking process geometries, design complexities make it increasingly difficult to continue to use traditional HDL design flows. Although HDL languages and tools have evolved, design cycles remain undesirably long.

Challenging applications are encountered today in all domains: advanced driver-assistance systems (ADAS), 5G, deep learning, computer vision, financial applications, aerospace and defense, etc. All of these require several processing kernels, often with complex control structures. This results in processing pipelines that are not easy to implement. Exploring architectural implementations to find the best strategy to meet constraints takes time and is costly.

Moreover, project schedule pressure as well as design costs increase the need to get the implementation right the first time. When bottlenecks are encountered in the late stages of development, refactoring portions of the design by hand can make it virtually impossible to stay on track.

Classical FPGA design flow also suffers when requirements change. It is indeed common that during product integration and validation phases, late specification changes come up or new constraints have to be applied. Simple changes in functionality or performance can require extensive design modifications. These design modifications can mandate an update to the architecture, leading to a rewrite of a portion of the low-level HDL implementation. This, of course, leads to potentially long re-validation cycles.

Benefits and challenges of HLS to speedup FPGA design

To help address the issue, high-level synthesis (HLS) compilers have emerged to enable designers to move to a higher level of abstraction. These are especially useful for highly complex portions of a design that can be easily expressed by higher-level languages, such as C or C++. The higher abstraction level makes it easier to describe complex algorithms and adapt quickly to functionality changes and is better suited for design re-use.

In addition to speeding up HDL designs, HLS also has the main advantage of easing design validation. With HLS, it is guaranteed that the high-level C/C++ description will be correctly implemented in gates. It is not necessary to make extended and time-consuming bit-level simulations. Moreover, since the high-level C/C++ code taken by HLS is executable, it is also used for high-level application validation.

Another key benefit of HLS is the ability to quickly explore different architectural design spaces by guiding the compiler with directives, often in the form of pragmas. This capability comes at a cost; to guide the compiler, the designer must first possess a deep understanding of the code and how data moves throughout the functions and loops. HLS compilers are static tools and do not offer any help in understanding the code’s dynamic characteristics. Furthermore, HLS compiler behavior is often difficult to predict in terms of resulting performance and resource utilization. Designers must therefore manually explore the design space by applying various pragmas, with their associated parameter settings, to the appropriate sections of code until design targets are met.
2 Overview of SLX FPGA

Benefits of SLX FPGA

SLX FPGA helps to convert your C/C++ code into an FPGA more easily, more quickly, and with higher performance. Leveraging standard high-level synthesis (HLS) tools from FPGA vendors, SLX FPGA tackles the challenges associated with the HLS design flow, including identifying non-synthesizable C/C++ code and non-hardware aware C/C++ code, detecting application parallelism, inserting pragmas, and determining optimal SW/HW partitioning.

SLX FPGA is a tool that sits between the source code and the HLS compiler. It performs dynamic analysis to extract deep insight into the behavior of the code. The analysis tools are then able to provide several perspectives on the code which can be especially useful when trying to understand performance blockers. It analyzes parallelism in the code as well as memory consumption and memory accesses. It can evaluate the impact of each section of code on the performance of the overall program execution. This makes it easier for designers to determine if and where the code needs refactoring. The tool then takes this one very important step further and uses those insights to determine what pragmas to use, where to use them, and with what parameters in order to achieve the most optimal implementation possible. Moreover, SLX FPGA can keep track of resource consumption as it explores the vast design space to keep the final implementation within the constraints specified by the designer. It does all of this automatically, removing the need for manual, iterative architectural exploration.

SLX FPGA Flow

*Figure 1* gives an overview of the SLX FPGA workflow for implementation and optimization of an HLS application, starting with a C\C++ specification.

- **Synthesizability refactoring**: SLX performs automatic source code refactoring of some commonly occurring synthesizability issues. For instances where automatic refactoring is not possible, detailed hints with examples are provided that facilitate refactoring.
- **Parallelism detection**: SLX detects parallelization patterns for loops. Currently, two parallelization patterns are supported: (1) data-level parallelism and (2) pipeline-level parallelism. In addition, parallelization blockers, such as loop carried dependencies, are reported.
- **Application analysis**: SLX’s analysis features are designed to help HLS users focus on the most important hotspots and bottlenecks of the system. Automatic hotspot detection at the granularity level of a loop. Read/write counts for variables are reported per access location.
- **Hardware optimization**: SLX selects the appropriate optimization for a target FPGA. For example, whether to pipeline or unroll loops and with what unroll factor, considering the available resources and interface constraints. Also, other critical decisions like array partitioning, array reshaping, and function inlining are done here.
- **HLS pragma insertion**: SLX automatically inserts HLS pragmas here. Developers have the option to review and edit the generated pragmas.
3 Use case: An industrial object tracking application

Our use case is an industrial image processing application that divides the input image into blocks and then detects and tracks features in each block. Figure 2 provides an overview of the application. The main processing steps of the algorithm are:

- Extracting blocks from images,
- Extracting pixels from parameterized regions of the blocks,
- Computing aggregated values per block, and
- Combining linear parameters.

The application is targeted towards small low-cost FPGAs and the on-chip memory size needs to be minimized. The original algorithm, written in C, is a good candidate for HLS. The initial HD image is divided into separated blocks, leading to a lot of parallelism available, which is common in most image processing algorithms. The algorithm uses integer arithmetic operations and fixed-point arithmetic.
3.1 HDL Implementation

The algorithm has been implemented in structural Verilog using a stream-oriented dataflow control. The entire algorithm is made of processing stages connected by FIFOs. Each stage processes incoming pixels and metadata before sending them to the next stage. The processing is heavily pipelined; long combinatorial paths are broken up into stages, and FIFOs are used to synchronize between them. The IP is able to process pixels at 100 Mpix/sec at 100 MHz and can sustain a very high frame rate, up to 80 fps.

The projected deployment volume for the IP was high; therefore, thorough testing was required to minimize the risk of bugs. For testing and validation, a dedicated validation framework was designed. This validation framework made an automated comparison of the outputs of the C reference implementation with the output of the Verilog simulator. A suite of test scenarios was also developed to cover all of the corner cases that may be encountered by the application. The HDL flow took almost two months for the first validated hardware implementation. Moreover, during the course of the project, the specifications were changed several times. These specification changes not only required modifications in the IP, but the testing and validation framework had to be updated as well.

![Figure 3: SLX hints view](image)

3.2 Initial HLS Implementation

In order to evaluate the SLX FPGA, a new C/C++ implementation of the image processing algorithm was developed by an engineer who is an expert in FPGA design. This is a raster scan dataflow implementation of the algorithm, similar to the HDL implementation. Rather than storing the complete image in a DDR frame buffer, pixels are processed line by line as they arrive on the input
video stream. This approach allows us to avoid the cost and energy dissipation associated with large external DDR memories. Implementing with HLS was very quick; getting the code to be synthesizable took less than a day (SLX FPGA helped with some suggestions). Validation was done at the C-code level and took a minimal amount of time. It is worthwhile pointing out here that the specification updates to the original C algorithm that we had to translate to RTL and validate would have been much quicker in the HLS flow. However, our initial HLS implementation did not meet our performance goals. Therefore, it was essential to analyze the code to understand why and what we could do about it. This is where SLX FPGA provided deep insights on the code.

3.3 Analysis & Optimization with SLX

SLX provides many analysis features that help understand the hotspots and performance bottlenecks of the application. Figure 3 shows the SLX hints view. With SLX’s automatic hotspot detection, the first thing we notice is that the loop at row 24 takes more than 96% of the execution time. This is the loop that maps incoming pixels to blocks. The next step is pinpointing the bottleneck in this loop. For this, we look at the memory analysis view (Figure 4). We see that a variable bb_lookups is accessed 6.2 billion times; this is way too high for an HD resolution image. Moreover, the HLS compiler will map the variable to external memory by default (due to its size). To improve this, we modified the code to only check a local subset of blocks close to the block of the current pixel being processed. Figure 6 shows the pseudo-code of how the application is modified to search pixels in blocks. The access count to the bb_lookups goes down to 170 million (from 6.2 billion), a 97% reduction, and the execution time percentage is reduced significantly, as shown in Figure 5. (Notice how the execution percentages went down.) We were able to quickly test that the code changes did not impact functionality.

Figure 4: SLX Memory analysis view

Figure 4 also shows other potential memory bottlenecks for this application (i.e., larger-sized variables with high access counts), ‘src_image_pixels’, ‘acc_lpix’, and ‘acc_ltot’. Figure 5 shows that
loop-carried dependencies (LCDs) are detected on two of these variables. LCDs play a very important role in unrolling or pipelining a system of loops.

LCDs form a feedback loop when synthesized. If the variable with LCD is mapped to an external single-port memory it takes at least two cycles for a loop iteration, e.g., one for reading the variable from the memory and one for writing it back; the next loop iteration can’t start before this process completes. This hampers speed-up gains with parallelization. However, if the variable is mapped onto a local register and the combinatorial logic path between the read and write operations is short enough, the whole process can be completed within a single cycle. For this application, we copied a subset of these variables to smaller local variables related to the local blocks so that they are synthesized as registers. Figure 6 shows an example of how the pipelining performance of a loop is improved with this optimization, i.e., by collecting inner loop aggregates in local registers and adding them up outside the loop. This increases the pipeline throughput to 1 pixel/cycle. SLX provides a high-level approach to hone-in on the variables that matter most, i.e., the loop-carried-dependencies in the hotspot loops of the application.
3.4 HLSPragma insertion

Our final optimization step with this application is to guide HLS in terms of how it should implement the design. Designers generally have to figure out when and how to partition and/or reshape arrays, whether to pipeline or unroll each loop (and for unroll, decide what the correct unroll factor is), and so on. SLX's optimization algorithm uses internal models along with the information collected through static and dynamic analysis to perform an exhaustive design space exploration. The goal is to minimize latency while meeting the area constraints of the target device or other user-defined constraints. Once the best set of pragmas for the application is found, SLX will show the recommended pragma settings for designer approval. Figure 7 shows the SLX code transformation wizard, where the pragmas that will be inserted can be selected. For this case study, we inserted all pragmas automatically selected by SLX. The next section compares the results of the different versions of this application.
Figure 7: SLX Code Transformation Wizard for automatic pragma insertion

4 Results

Figure 8 shows a bar graph of latencies of our different implementations on a Xilinx Ultrascale+ target. The second bar on the graph is our initial HLS implementation; the latency for this version is almost 90 times more than our HDL implementation. The third bar is the result of using SLX to find the critical hotspots and memory bottlenecks and the resulting code refactoring. We already see a 20x improvement over the initial HLS implementation; however, this is still slower than our HDL implementation. For the fourth and final version, we used the SLX automated optimization to generate HLS pragmas to the refactored code. This version even outperforms our HDL version, with a reduction in latency of 39%.

In terms of resources, our final implementation uses 2.4x more LUTs, almost the same number of flip-flops, 50% more BRAM, and 20% more DSP blocks compared to our handwritten HDL implementation. However, the design still fits in our target FPGA; therefore, we consider this a favorable tradeoff for the lower latency.

1 SLX version 2020.2 is used for the results presented in this paper.
Figure 8: Comparison of latency for the different synthesized versions

Figure 9 shows a timeline of the project with the various design flows. The traditional HDL design flow took almost two months to get the first validated hardware; where four weeks were spent in the initial design and coding in Verilog, two weeks for architectural refinements and optimization and another two weeks for testing and validation. The traditional HLS flow, without SLX FPGA, took almost three weeks, where most of the time was spent in architectural exploration and optimization. With SLX, we were able to get through the whole cycle in less than a week. Most importantly, specification updates can be implemented much faster with the SLX flow than in HDL or even the traditional HLS flow.

Figure 9: Project timelines comparing different development flows
5 Conclusions

We used SLX FPGA as an analysis tool to drive our code refactoring effort to optimize an industrial image processing application. We then used SLX FPGA to automatically explore the design space and find the optimal pragmas and pragma settings to minimize latency while staying within our resource utilization constraints. The results are compared to a handwritten HDL implementation. By using SLX FPGA and Vivado HLS, we were able to beat the performance of our HDL, with a speed-up of 64%. The real benefit is that it took us only one week to implement, optimize, and validate the C application. The flow not only yielded better results in a fraction of the time, but it is also far less impacted by last-minute changes to the algorithm when compared to the HDL flow.