Using SLX FPGA for performance optimization of SHA-3 for HLS

Date	2019-09-19
Author	Zubair Wadood

SILEXICA

Abstract

SLX FPGA facilitates converting your C/C++ project into an FPGA bitstream easier and with higher performance. Leveraging standard HLS (High Level Synthesis) tools from FPGA vendors, SLX FPGA tackles the challenges associated with the HLS design flow. In this paper, the results of an SLX FPGA optimized implementation of a Secure Hash Algorithm (SHA-3; also known as Keccak) are compared to a competition-winning hand-optimized HLS implementation of the same algorithm. **SLX provides a nearly 400x speed-up over the unoptimized implementation and even outperforms the hand optimized implementation by 14%.** Moreover, it is also more resource efficient, consuming nearly 3.6 times less look-up tables and 1.76 times less flip-flops.

1 Introduction

Heterogeneous computing with FPGAs has emerged as an attractive option to implement high-throughput and energy efficient designs. FPGAs are increasingly being used to accelerate time-critical functionalities for various applications. With a compound annual growth rate of 8.5%, the FPGA market is one of the fastest growing markets in the tech industry[1]. As FPGAs find their way into diverse application domains, new methodologies for synthesis are required to address the challenges of using FPGAs and lower the bar on the expertise required for designing FPGA-based systems. More recently, high-level synthesis (HLS) tools that transform C/C++ designs into FPGA bitstreams are becoming mainstream.

HLS tools improve design and verification productivity, helping developers address the issues of increasing time-to-market due to the rising complexity of FPGA-based systems. However, exploiting the computational power of an FPGA using HLS tools remains a challenge. Non-trivial trade-offs and design decisions must be made. These decisions not only require deep insights of the application but also a significant knowledge of the FPGA architectures. SLX FPGA uses a unique combination of state-of-the-art static and dynamic analysis techniques to extract insights from an application. Its advanced optimization heuristics use the insights for making trade-offs that could otherwise require several design cycles by an HLS expert. SLX FPGA not only reduces the time and effort for optimizing applications for FPGAs, but also makes it easier for non-experts to use HLS techniques effectively, enabling software engineers with little hardware knowledge to optimize applications for high-level synthesis.

Secure hash algorithms (SHA) are at the heart of security and integrity protection infrastructure for communication systems and databases. SHA-3 (also known as Keccak) is the latest member of the SHA family recognized by the National Institute of Standards and Technology (NIST). In 2015, a contest was held to find the best HLS implementation of Keccak at the International Symposium on Applied Reconfigurable Computing. The implementation by Ekawat and Homsirikamol[2] won this competition. SLX FPGA is used here to optimize an implementation of the Keccak algorithm (without HLS pragmas). In this paper, we compare the results of the SLX optimized implementation to the competition winning hand-optimized version.

The first section provides an overview of SLX FPGA followed by an elaboration of using SLX to optimize an implementation of Keccak for high-level synthesis. The last two sections present the results followed by a concluding summary.

[1] https://www.marketsandmarkets.com/Market-Reports/fpga-market-194123367.html, 26 August 2019

[2] https://cryptography.gmu.edu/athena/sources/2015_04_15/Keccak/HLS_Keccak.zip, 26 August 2019

2 SLX FPGA Overview

One of the biggest challenges with using the HLS design flow is performance and design efficiency in terms of latency and area of the synthesized design. Improving the performance and area utilization of an HLS design requires parallelization and hardware-aware optimization. HLS design suits, such as, the Xilinx Vivado™ provide several mechanisms for implementing parallelization and hardware optimizations but depend on the developer to specify them through HLS pragmas. Specifying these pragmas often involve making sophisticated tradeoffs that are not trivial for large complex applications. SLX employs extensive application analysis techniques to acquire deep insights into an application. These insights are then used to reason about parallelization and optimization decisions.

Most HLS tools on the market rely solely on compiler-based static analysis. However, large parts of applications can be data driven; it is often not possible for static analysis tools to extract exact metrics for these parts. For example, with static analysis, the micro-operations in a block of code are known, but its execution frequency is only known at runtime. Moreover, static analysis often fails to provide precise memory access patterns: for example, failing to see through pointer chains and detecting underlying objects or function calls. SLX uses dynamic analysis to capture such essential information when it is missing from the static analysis results. In fact, SLX FPGA is the first and (to date) only commercially available tool that applies the power of dynamic analysis to HLS optimization.

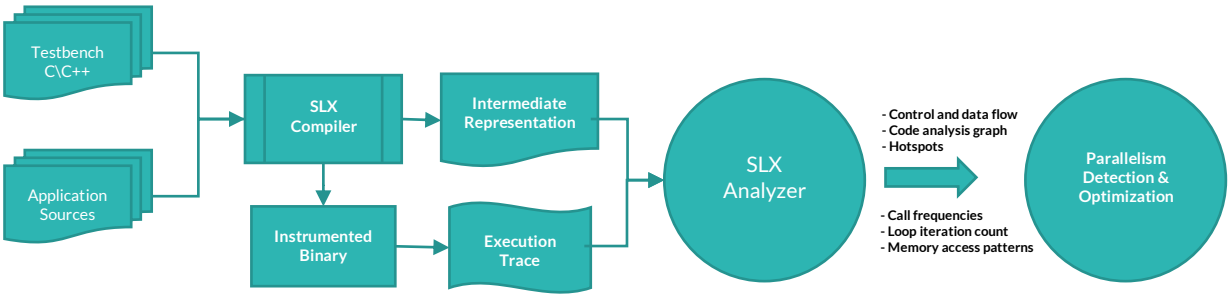


Figure 1: SLX Application analysis overview

Figure 1 provides an overview of SLX application analysis. The proprietary compiler of SLX instruments the application, when required, at the level of instructions and memory accesses. A trace is generated by executing this instrumented binary. Critical information, such as memory access patterns, loop trip-counts, call and conditional code execution frequencies are extracted from this trace. State-of-the-art analysis techniques are used to extract weighted code analysis graphs from the application, which is essential in reasoning about the different optimizations and to relate them back to the source files for code generation.

Another challenge for HLS users is synthesizability refactoring, especially for legacy applications. SLX FPGA facilitates user with automated and guided synthesizability refactoring. The process with

which SLX facilitates developers to transform their C/C++ application code into an optimized, HLS ready code, is divided into four steps. Figure 2 gives an overview of these steps.

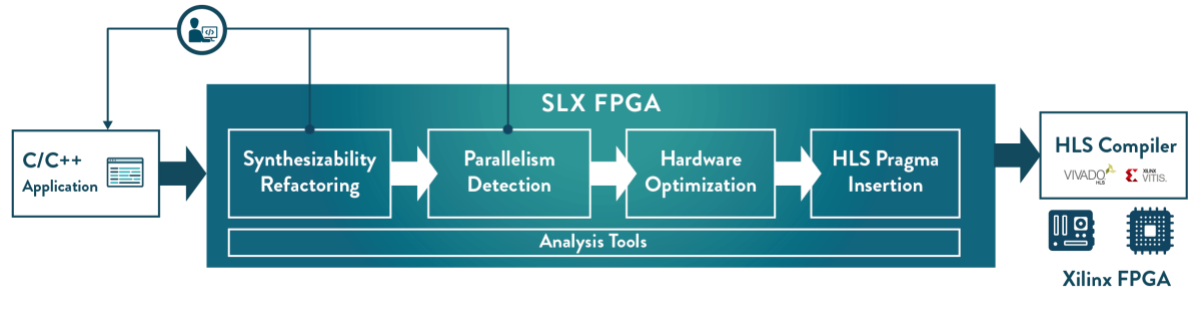


Figure 2: SLX FPGA workflow, starting from a generic C/C++ code to a HLS ready code optimized for HW implementation.

2.1 Synthesizability Refactoring

Current HLS compilers support a restricted set of the C/C++ language. Example restrictions include the use of dynamic memory, variable sized function arguments, function overloading, function pointers and recursion. For novice users, patching these synthesizability issues often require searching through thousands of pages of documentation. This makes synthesizing legacy software applications very cumbersome.

The SLX tool helps programmers with automated and guided refactoring of non-synthesizable code. For example, it automatically replaces non-synthesizable library function calls with synthesizable ones. Detailed hints are generated to guide the developer in transforming non-synthesizable pieces of code into synthesizable code.

2.2 Finding Parallelism

The performance of an FPGA implementation relies heavily on the degree of parallelism present in the application. HLS compilers, such as Vivado HLS, support several different constructs for implementing parallelism in hardware, e.g. pipelining, unrolling and dataflow parallelism. However, these HLS compilers rely on developers to identify parallelization opportunities and specify their properties using pragmas. This is a non-trivial task that requires not only a deep understanding of the application but also awareness of the underlying FPGA architecture. In FPGAs, exploiting parallelization translates to an increased resource consumption; therefore, a system level approach is required to target resources toward the most important performance bottlenecks of the system. For medium-sized to large applications, where every loop has several possible parallelization candidates, the complexity can build up quickly.

SLX FPGA currently supports two parallelization patterns: (1) data-level parallelism and (2) pipeline level parallelism. Data-level parallelism is selected for loops in which every computation in the for loop is either independent or can be mapped efficiently to a highly parallel hardware architecture (e.g., a multiply-add tree). Pipeline-level parallelism is selected for loops where different operations

within a single loop iteration can be parallelized, and where several loop iterations can execute simultaneously in individual pipeline stages[3].

2.3 Hardware Optimization

The next step in the SLX workflow is exploring and selecting hardware optimizations. The selected hardware optimizations are then passed onto the HLS compiler for implementation, in the form of HLS pragmas. Currently, SLX supports five types of hardware optimizations: (1) loop unrolling (2) loop pipelining, (3) array partitioning and interface design, (4) function inlining and (5) loop trip count generation.

2.3.1 Loop unrolling

If loop unrolling is detected as a feasible parallelization candidate for a loop, it is further evaluated for speed-up yields for the application on the target FPGA platform. Moreover, the optimal (hardware aware) unroll factors are calculated with respect to the available resources on the platform.

SLX FPGA takes a system level approach to parallelism. Utilizing the information collected in the static and dynamic analysis phases, SLX identifies the most critical bottlenecks and hotspots of the application. Cost-benefit analysis is conducted for different parallelization candidates and their various implementation variations, thus ensuring that the limited resources on an FPGA are utilized efficiently.

2.3.2 Loop pipelining

If loop pipelining is selected as a feasible parallelization candidate, then application speed-up from implementing this pipeline is estimated for the target FPGA. The final decision for the adaption of the parallelization candidate is made on bases of these estimates. Furthermore, optimal pipeline stages and initiation intervals for the loop on a given target platform are calculated.

During the step for finding parallelism, parallelization opportunities in sequential code are explored: i.e., feasible parallelization candidates are found. In the hardware optimization step, a cost benefit analysis determines if a parallelization candidate yields a significant speed-up for the application on a target platform and its fine-tuning parameters are calculated. In the final step, corresponding pragmas for these parallelization candidates are generated.

2.3.3 Array partitioning and interface design

Data-intensive applications are often constrained by communication and/or memory access bandwidth. To address these issues, HLS compilers support several array partitioning and interface

[3] Michael McCool, James Reinders, and Arch Robison. 2012. Structured Parallel Programming: Patterns for Efficient Computation (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

design options but rely on developers to select the right options and specify them through pragmas. Different interface types and array partitioning implementations have different area, bandwidth and access characteristics. Therefore, making these design decisions involve complex trade-offs. Furthermore, since more parallelism require more bandwidth, the choice of the right interface type or array partitioning is tightly coupled with the selected parallelization patterns for associated computational elements.

2.3.4 Function inlining

In HLS, C/C++ functions are normally synthesized as a separate hierarchy, i.e., all calling instances of these functions would use the same synthesized module, unless specified otherwise. Inlining a function forces its functionality to be synthesized in all calling functions. This has serious consequences for performance and area of the design. On one hand, non-inlined functions with multiple callers and/or those that are called from within parallelized loops often become performance bottlenecks. On the other hand, inlining functions with many instances could lead to a drastic increase in the area of the design. Therefore, making this decision manually is difficult. SLX optimization heuristics automatically inline functions where beneficial, considering a number or parameters such as calling frequency, degree of parallelization and implementation area estimates.

2.3.5 Trip-counts

Trip-counts help the HLS compiler optimize loops and generate latency reports for loops. SLX uses profiling information from realistic workloads to calculate trip-counts, and automatically annotates them to the target applications, saving the designer to go through this time consuming and error prone process.

2.4 Pragma Insertion

After SLX FPGA has determined the optimal pragmas that need to be inserted in the application, and their corresponding parameterization, a code generation process takes place. During this process, SLX automatically inserts the calculated annotations in the source code, in order to direct the HLS compiler towards the optimal solution. SLX presents a preview of the code that will be generated, side by side with the original source code, for the user to perform final tuning. Once the user is satisfied, SLX generates the annotated code, create a Vivado project, and call the HLS compiler to obtain the actual synthesis results for the hardware IP block. This seamless integration enables the user to import the generated project inside Xilinx's tools, and continue inside a familiar development flow.

3 Using SLX for the Optimization of Keccak

SLX offers a useful set of visualization capabilities to help understand how the application executes. Figure 3 displays the analysis graph generated for Keccak, which is an abstract representation of SLX analysis data. The rectangular nodes (in the first two rows) represent functions and the square nodes represent global data variables. The function nodes are annotated with their respective (software) execution times. The self-cost is the total time spent within the function itself, whereas the total cost includes the self-cost along with the cost of all the functions it calls. The size of the square boxes represent the relative sizes of the data variables. Selecting a data node highlights its accessing functions and read/write relationships; selecting a function node highlights the call paths

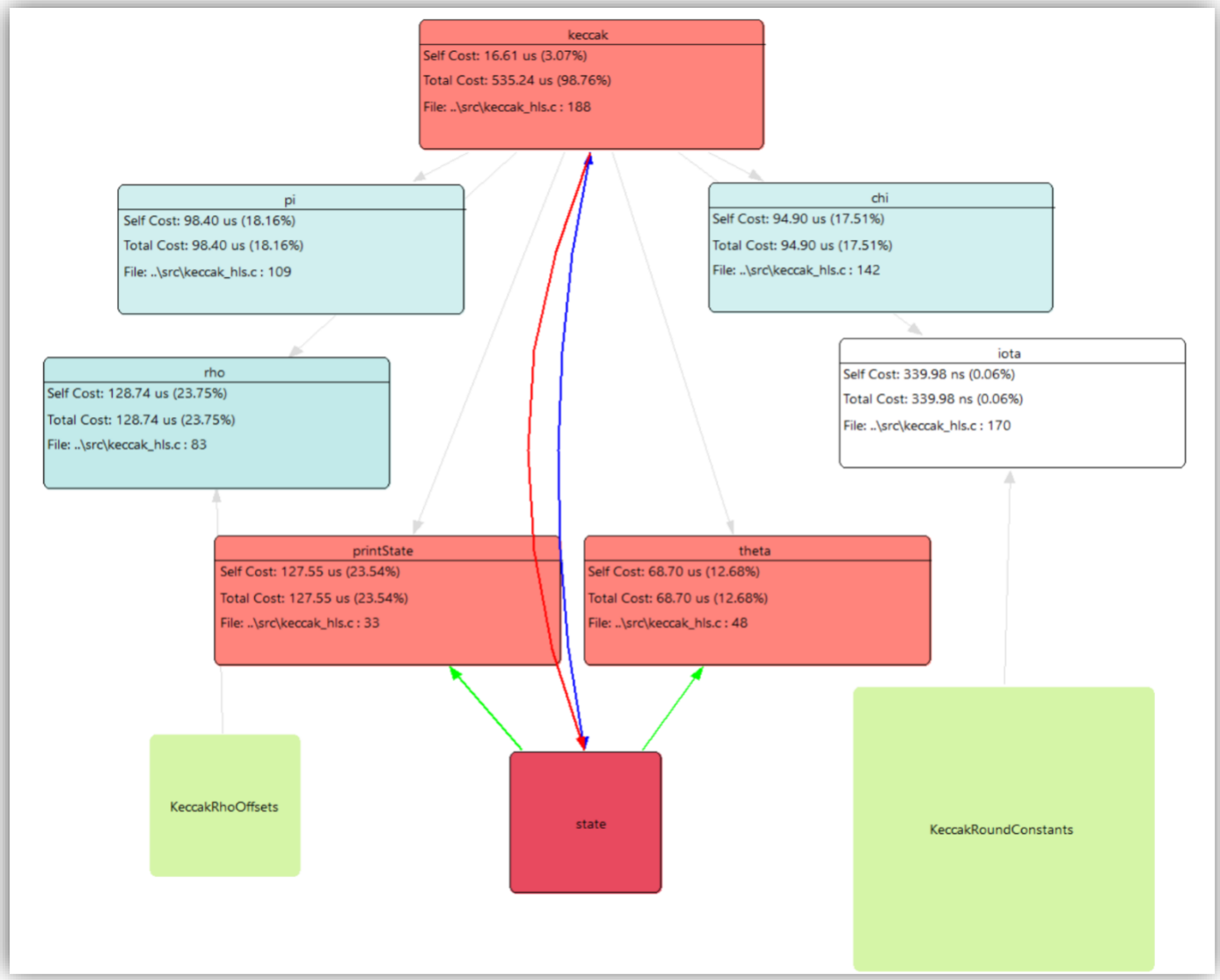


Figure 3: Analysis graph for Keccak generated by SLX. The rectangular nodes represent functions and the square nodes represent global variables. The arrows between the nodes represent caller-callee or data access relations

that reach that function. The analysis graph can easily be configured to represent other aspects of the application, e.g., local variables, read/write access counts, threads and processes. For Keccak, we see that the 'rho' function consumes the most time in software execution, and only access the 'KeccakRhoOffsets' global variable. The 'KeccakRoundConstants' is the largest of global arrays but is only accessed by the 'iota' function which is not computationally extensive. The state variable is accessed from three different functions. Developers can often use such insights into the application to manually fine tune optimization and/or manually transform the application to address bottlenecks.

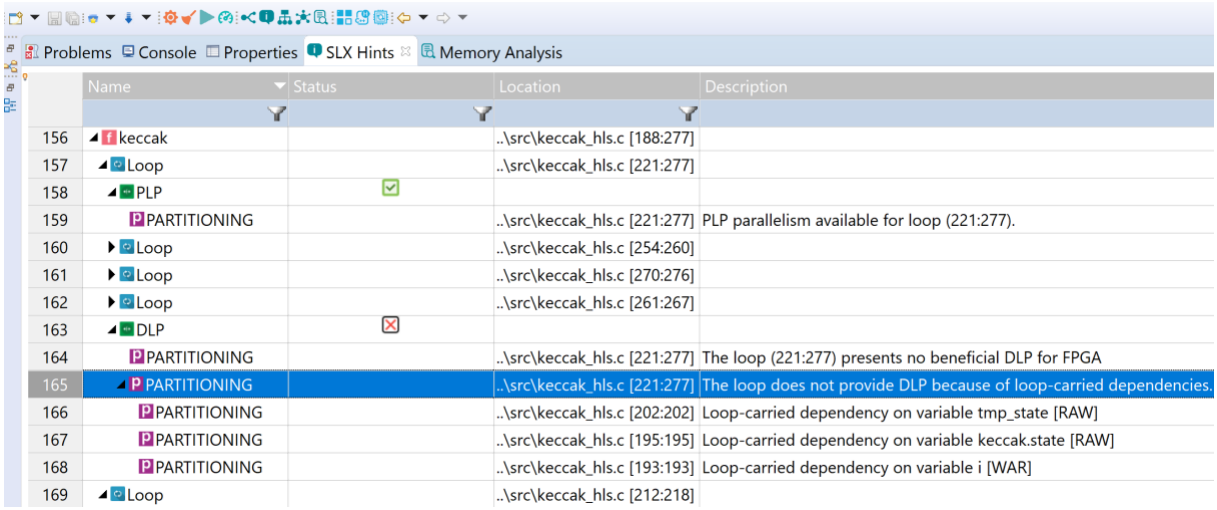
This section outlines how SLX FPGA is used to optimize an implementation of Keccak, illustrating different phases step-by-step.

3.1 Synthesizability Refactoring

The implementation of Keccak used for reference did not have any synthesizability issues. Therefore, refactoring was not required.

3.2 Parallelism Detection

Figure 4 shows the results of SLX parallelization tools; a total of 43 possibilities of applying DLP and PLP parallelization patterns on different loops are explored. 31 of these parallelization opportunities are feasible and 12 are not feasible (not possible to realize, e.g. due to data dependencies). For unfeasible parallelization opportunities, SLX provides detailed information on why a parallelization pattern cannot be applied. For example, in **Figure 4**, the loop on lines [221:277], a DLP cannot be implemented due to loop carried dependencies on three variables. In some cases, developers can use this information to modify their implementation and expose more parallelism in the algorithm, e.g., by adding some synchronization mechanisms.



	Name	Status	Location	Description
156	keccak		..\src\keccak_hls.c [188:277]	
157	Loop		..\src\keccak_hls.c [221:277]	
158	PLP	✓		
159	PARTITIONING		..\src\keccak_hls.c [221:277]	PLP parallelism available for loop (221:277).
160	Loop		..\src\keccak_hls.c [254:260]	
161	Loop		..\src\keccak_hls.c [270:276]	
162	Loop		..\src\keccak_hls.c [261:267]	
163	DLP	✗		
164	PARTITIONING		..\src\keccak_hls.c [221:277]	The loop (221:277) presents no beneficial DLP for FPGA
165	PARTITIONING		..\src\keccak_hls.c [221:277]	The loop does not provide DLP because of loop-carried dependencies.
166	PARTITIONING		..\src\keccak_hls.c [202:202]	Loop-carried dependency on variable tmp_state [RAW]
167	PARTITIONING		..\src\keccak_hls.c [195:195]	Loop-carried dependency on variable keccak.state [RAW]
168	PARTITIONING		..\src\keccak_hls.c [193:193]	Loop-carried dependency on variable i [WAR]
169	Loop		..\src\keccak_hls.c [212:218]	

Figure 4: Results of SLX parallelization tools

It is worth noting that the parallelism detection algorithms in SLX consider that the application computation will be mapped into reconfigurable hardware, thus expanding the set of detected parallelism patterns over traditional parallelizing compilers targeted at accelerating code running on conventional processors. For example, pipelining the loop [221:277] in **Figure 4** in the presence of loop carried dependencies yields a benefit on an FPGA and guarantees correct results by synthesizing dependency forwarding mechanisms. Whereas on a conventional processor, pipelining the same loop would cause a race condition unless additional protection mechanisms are used. SLX parallelization heuristics are hardware-aware and therefore able to detect such instances.

3.3 HW Optimization

Figure 5 shows a code snippet of the 'rho' function from the Keccak application after pragma insertion. It can be seen that one of the loops is parallelized using the unroll pragma. However, simply unrolling the function does not give much speed-up since it has a bottleneck on memory accesses to the 'KeccakRhoOffsets' array. Therefore, SLX partitions this array, eliminating the bottleneck. Also note that the function is selected for inlining. This is due to the fact that it has only one calling instance in the application and creating a separate hierarchy for it is simply an overhead.

```
void rho(UINT64 A[25])
{
#pragma HLS inline
  static const unsigned int KeccakRhoOffsets[25] = {
....
  for(x=0; x<5; x++) {
#pragma HLS loop_tripcount min=5 max=5
#pragma HLS array_partition variable=KeccakRhoOffsets complete
    for(y=0; y<5; y++) {
#pragma HLS loop_tripcount min=5 max=5
#pragma HLS unroll factor=5 skip_exit_check
      A[index(x, y)] = ROL64(A[index(x, y)],
                             KeccakRhoOffsets[index(x, y)]);
    }
  }
}
```

Figure 5: Code snippet illustrating HW optimization for the 'rho' function in Keccak

3.4 Pragma insertion

SLX automatically generates pragmas that implement the optimizations mentioned in the previous sections. **Figure 6** shows the results of pragma generation for the Keccak application. The upper section is used to manually select/de-select the pragmas to generate. The generated code preserves line-to-line mapping; the lower section is used to compare the generated code against the original code.

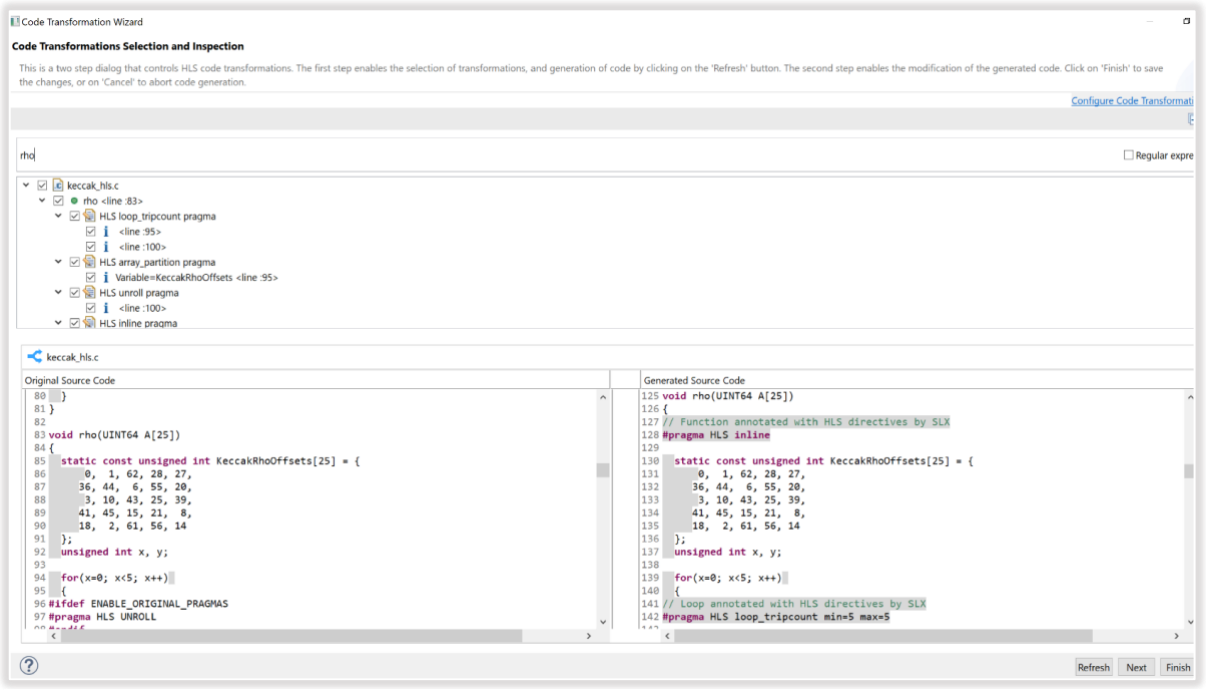


Figure 6: SLX HLS pragma generation

4 Results

Three implementations for the Keccak on the Zynq Ultrascale+ ZCU102 platform are synthesized: (1) an unoptimized software implementation (without HLS pragmas) synthesized directly with Vivado HLS, (2) an SLX optimized implementation and (3) the hand optimized implementation by Ekawat et al.[2]. **Table 1** shows the latency results of the three implementations. SLX provides a 397x speed-up over the unoptimized implementation and even outperforms the hand optimized by approximately 14%. This is because the hand-optimized missed some array partitioning and function inlining opportunities. SLX takes a systematic approach to these optimizations and is less likely to miss such optimization opportunities.

	Latency (Clock Cycles)	Speed-up
Unoptimized	17076	1
SLX optimized	43	397
Hand optimized	49	348

Table 1: Performance results of different Keccak implementations

Table 2 summarizes the resource utilization estimates for the three implementations. It is observed that the SLX optimized implementation is the most resource efficient. The hand-optimized implementation consumes nearly 3.6 times more look-up tables and 1.76 times more flip-flops. Thus, the SLX optimized implementation not only outperforms the hand-optimized implementation but does so with much fewer resources.

	Unoptimized	SLX optimized	Hand optimized
BRAM_18K	18	2	2
DSP48E	0	0	0
FF	2025	2587	4548
LUT	4504	14288	50033

Table 2: Resource utilization for Keccak implementations

[2] https://cryptography.gmu.edu/athena/sources/2015_04_15/Keccak/HLS_Keccak.zip, 26 August 2019

5 Conclusions

Inserting pragmas that drive HLS compiler optimizations involves complex trade-offs and a system level approach is required to reach optimal results. SLX utilizes state-of-the-art static and dynamic analysis techniques coupled with advanced parallelization techniques, as well as optimization heuristics for system level optimization of HLS applications. An SLX optimized implementation of a secure hash function (Keccak) is compared with a competition-winning hand-optimized implementation and reveals that the SLX optimized implementation not only outperforms the hand optimized implementation but does so consuming fewer resources. Furthermore, SLX FPGA also performed this analysis and optimization in a matter of hours, reducing the effort from weeks or months.

References

[1] https://www.marketsandmarkets.com/Market-Reports/fpga-market-194123367.html, 26 August 2019.

[2] https://cryptography.gmu.edu/athena/sources/2015_04_15/Keccak/HLS_Keccak.zip, 26 August 2019.

[3] Michael McCool, James Reinders, and Arch Robison. 2012. Structured Parallel Programming: Patterns for Efficient Computation (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

About the author

Dr. Zubair Wadood

Zubair is a technical marketing engineer at Silexica GmbH. He completed his PhD in computer science from the University of Leuven, Belgium in 2014; his interests include embedded systems and high-performance computing. Before joining Silexica, he has worked with Mentor Graphics and u-blox.