Adaptive Beamformer: An HLS Optimization Case Study with SLX FPGA

Date	2020-04-09
Author	Zubair Wadood

SILEXICA

Abstract

SLX FPGA is a powerful tool from Silexica that provides a considerable productivity boost when using high-level synthesis (HLS) to implement FPGA applications in C/ C++, through automated analysis and optimization. SLX combines static and dynamic analysis techniques to gain deep insights into the application, which are then used to make various optimization decisions. This type of analysis often leads to increased performance and more efficient resource utilization than manually optimized designs. In this paper, the latency and utilization metrics of an adaptive beamforming algorithm optimized by SLX FPGA are compared to the metrics of the same algorithm hand- optimized by an HLS expert. SLX FPGA achieves a lower latency and cuts development time from weeks down to minutes, using an automated flow that eliminates the need for knowledge of the algorithm and target architecture.

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. 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.

Adaptive beamforming is a critical part of sensor array processing and is used in a wide array of applications, including modern radars and wireless communications infrastructure (such as 5G base stations). This paper examines the use of a C-based description of the Modified Gram-Schmidt QR decomposition (MGS-QRD) algorithm with weight back substitution (WBS) for implementation on a Xilinx FPGA[1]. This implementation is manually optimized for Xilinx FPGAs using HLS pragmas and synthesized using Vivado HLS. The resulting implementation outperforms its CPU counterparts in both performance and energy efficiency by a huge margin.

We use SLX FPGA to automatically generate HLS optimization pragmas for this adaptive beamforming algorithm and compare the results with the manually optimized implementation. The next section gives a brief overview of SLX FPGA, followed by a discussion on the results.

1 "Adaptive Beamforming for Radar - Xilinx." 24 Jun. 2014, https://www.xilinx.com/support/documentation/white_papers/wp452-adaptive-beamforming.pdf. Accessed 20 Mar. 2020.

2 Overview of SLX FPGA

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 tools, such as Xilinx Vivado™ HLS, provide several mechanisms for implementing parallelization and hardware optimizations but depend on the developer to specify them through HLS pragmas. Specifying these pragmas often involves making sophisticated tradeoffs that are not trivial for large or complex applications. SLX employs extensive application analysis techniques to acquire deep insights into an application. These insights are then used to reason through parallelization and optimization decisions. Figure 1 gives an overview of the different steps in the SLX FPGA workflow; the following subsections describe each of these steps in detail.

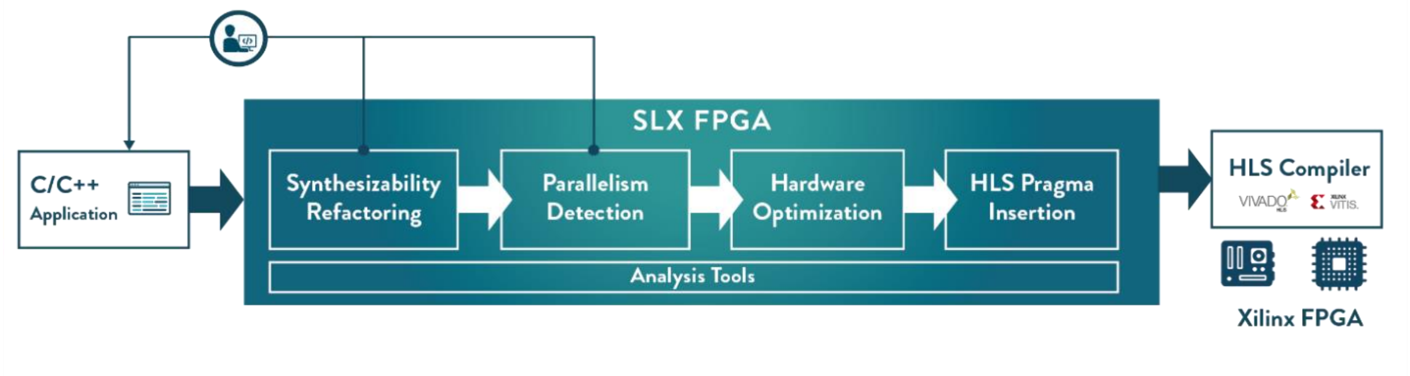
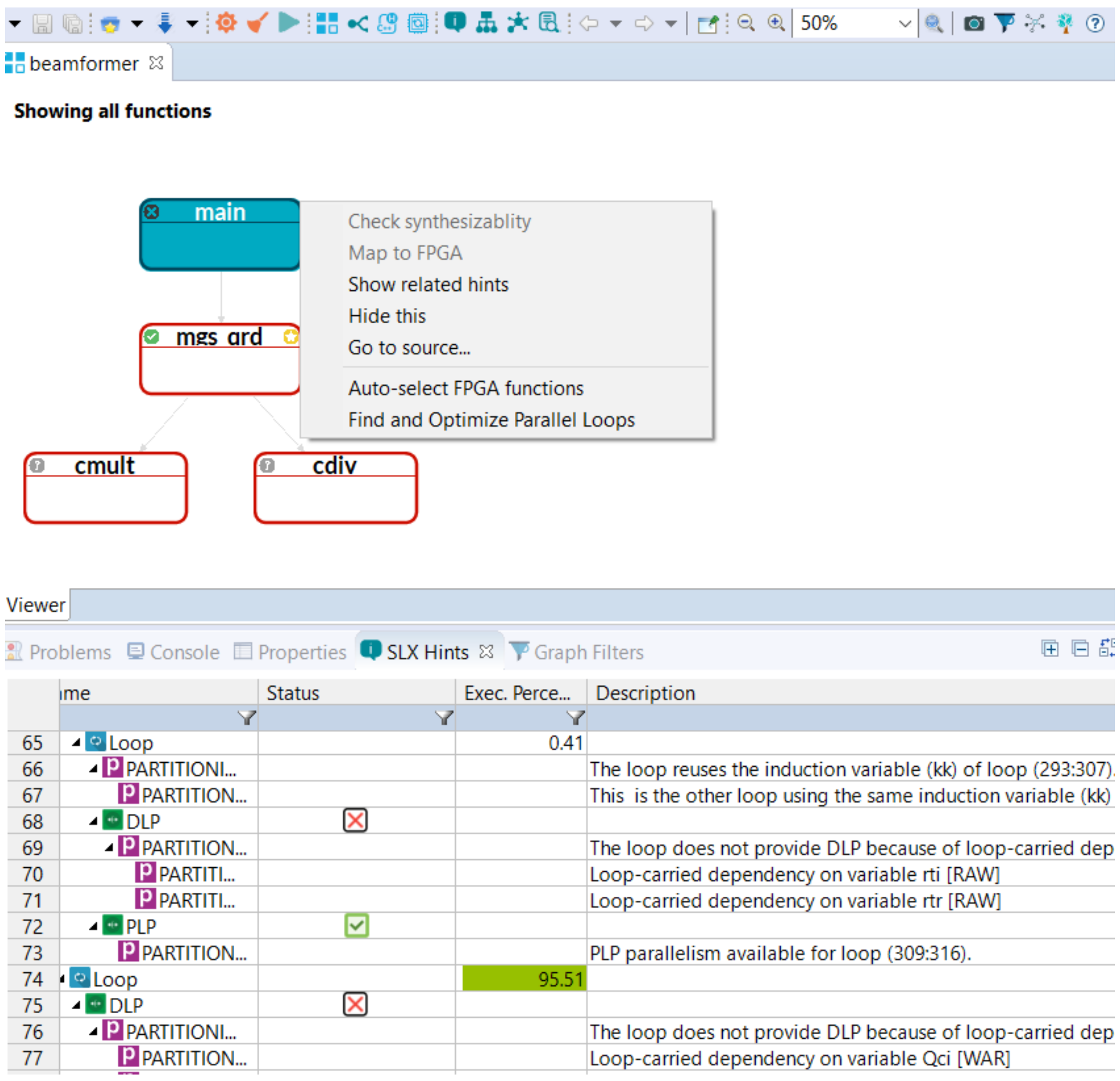


Figure 1: SLX FPGA workflow overview

2.1 Synthesizability Refactoring

Current HLS compilers support a restricted set of the C/C++ language and language constructs. Example restrictions include the use of dynamic memory, variable number of function arguments, function overloading, function pointers, and recursion. For novice users, patching these synthesizability issues often requires searching through hundreds 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 certain 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. SLX FPGA provides a graphical function mapping editor, where users can select a function to check for synthesizability.



	ıme	Status	Exec. Perce...	Description
65	Loop		0.41	
66	PARTITIONI...			The loop reuses the induction variable (kk) of loop (293:307).
67	PARTITION...			This is the other loop using the same induction variable (kk)
68	DLP	☒		
69	PARTITION...			The loop does not provide DLP because of loop-carried dep
70	PARTITI...			Loop-carried dependency on variable rti [RAW]
71	PARTITI...			Loop-carried dependency on variable rtr [RAW]
72	PLP	☑		
73	PARTITION...			PLP parallelism available for loop (309:316).
74	Loop		95.51	
75	DLP	☒		
76	PARTITIONI...			The loop does not provide DLP because of loop-carried dep
77	PARTITION...			Loop-carried dependency on variable Qci [WAR]

Figure 2: An SLX screenshot.

Figure 2 displays a sample screenshot of SLX FPGA with the Function Mapping Editor at the top and the SLX hints tab opened at the bottom. The upper half shows the function mapping editor, and the lower half shows the SLX Hints window. The Function Mapping Editor allows users to interactively select functions for synthesizability checking, mapping them to an FPGA and optimization, etc. The SLX hints window shows different parallelization hints, (i.e., whether a loop is a good candidate for unrolling or pipelining). The SLX hints window also shows execution percentages (% CPU time) and partitioning information, such as loop-carried dependencies, of each loop and function.

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 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 automatic parallelization of two patterns: (1) data-level parallelism and (2) pipeline-level parallelism. Data-level parallelism is selected for loops in which every computation in the 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.

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 reshaping, (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 the basis 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 reshaping

Data-intensive applications are often constrained by communication and/or memory access bandwidth. To address these issues, HLS compilers support several array partitioning, reshaping, and interface design options but rely on developers to determine and specify 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 involves complex trade-offs. Furthermore, since more parallelism often requires more bandwidth, the choice of the right interface type or array partitioning is tightly coupled with the selected parallelization patterns for associated computational elements and vice versa. Thus, SLX reshapes arrays on function interfaces to suit the bandwidth requirements of the underlying parallelized computational patterns and the constraints of the selected interfaces. Similarly, local arrays are partitioned to meet the bandwidth requirements of the parallelized loops that access them.

2.3.4 Function inlining

In Vivado 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-count pragmas tell the HLS compiler how many times a loop is run. This information is used by Vivado HLS to generate more accurate latency reports for loops. SLX uses profiling information from actual workloads to calculate trip-counts and automatically annotates them to the target applications, saving the designer from going 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, 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, creates a Vivado project, and calls 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 into Xilinx's tools and continue inside a single development flow.

3 Results and Discussion

The implementation of the MGS-QRD+WBS algorithm for radar beamforming optimized by an expert HLS user is compared to that of SLX. The application is a system of eighteen loops, four of which are nested at level 3, ten at level 2 and the remaining at level 1. There are six arrays being passed on the interface and twenty-four local arrays. Exploring optimal pipelining and unrolling for these loops along with the appropriate unroll factors and finding the right array partitioning and reshaping implementations for all these arrays is a formidable task. It would take an experienced HLS user with some domain knowledge at least 1-2 weeks to optimize this application. A novice user could take several months. *SLX takes less than* ***5 minutes*** *to analyze and optimize this application.*

Table 1 gives an overview of the different pragmas inserted in both versions of the application.

Pragma	SLX 20.1	Hand Optimized
Array Partitioning	15	23
Array Reshape	2	6
Loop Pipeline	3	9
Loop Unroll	11	12
Inline	2	0
Loop Trip-count	18	13
Total	47	63

Table 1: Pragmas inserted in the different versions of the application

Table 2 summarizes the latency comparison of different implementations; the upper bound acceptable latency for this algorithm is 3.5 ms. Without any HLS optimization pragmas, the latency is 14.75 ms, which is beyond the acceptable range. The hand-optimized version reports a latency of 1.94 ms (7.6x faster than the unoptimized implementation). In comparison, the SLX 2020.1-sp1 optimized implementation has a latency of 1.82 ms or a speed-up of 8.1 times over the unoptimized. *The SLX implementation outperforms the hand-optimized version in terms of latency.* (Note that at the time of this writing, SLX FPGA 2020.1-sp1 was still in beta and not publicly available. SLX FPGA 2020.1 results are shown in the table for reference).

	Latency (ms)	Speed-up
Unoptimized	14.75	1
SLX 20.1	2.08	7.0x
SLX 20.1-sp1	**1.82**	**8.1x**
Hand optimized	1.94	7.6x

Table 2: Latency results for a 30x30 cells MGS-QRD+WBS algorithm for ZynqUltraScale+ target (ZCU 102)

Table 3 summarizes the resource utilization of the different versions of the application. The unoptimized version required the minimum resources because it is not parallelized. The SLX version requires more resources than the hand-optimized one. This is because the internal optimization engine of SLX considers resources as constraints and the only objective is to reduce latency. However, users can further restrict the resources available for SLX to implement an IP block to meet their requirements.

	Unoptimized	SLX 20.1 optimized	SLX 20.1-sp1 optimized	Hand optimized	Total available on device
BRAM_18K	24	129	100	128	1824
DSP48E	24	644	928	376	2520
FF	5035	127159	148653	100071	548160
LUT	6860	156056	148564	107626	274080

Table 3: Resource utilization comparison for a 30x30 cells MGS-QRD+WBS algorithm for ZynqUltraScale+ target (ZCU 102)

4 Conclusion

We compare two versions of a core algorithm for adaptive beamforming. One version is manually optimized by a domain expert with considerable HLS experience. The other is automatically optimized with SLX FPGA. We estimate that an expert would require at least a week or more to analyze the application and implement optimization in the form of HLS pragmas. With SLX, this time is reduced to mere minutes. Moreover, no domain knowledge or HLS experience is required to implement these optimizations with SLX. In terms of performance, the SLX solution exceeds the speed-up obtained from manual-optimizations for a comparable cost of resources. SLX proves to be a lifesaver for developers who inherit legacy applications or new HLS users who could take months to get through the steep learning curve of HLS optimization. Experts benefit from it as well by gaining time through quick design space exploration for various optimizations and deep application analysis that helps open opportunities for further optimizations.