SILEXICA

SLX FPGA
Accelerate the journey from C/C++ to FPGA

Silexica Facts

RWTH AACHEN UNIVERSITY

Est. 2014 after a decade of research

Team of world leading software and hardware experts

60 people worldwide, engineering HQ in Germany

3 offices and worldwide local support engineers

MITSUBISHI HITACHI Inspire the Next HUAWEI FUJITSU MBDA DENSO THALES RICOH

GERMANY
USA
JAPAN

High-Level Synthesis Benefits

- High-level synthesis (HLS) provides C/C++ based FPGA design, benefits include
  - Faster implementation
  - Faster verification
  - Flexible design re-use
  - Faster design space exploration

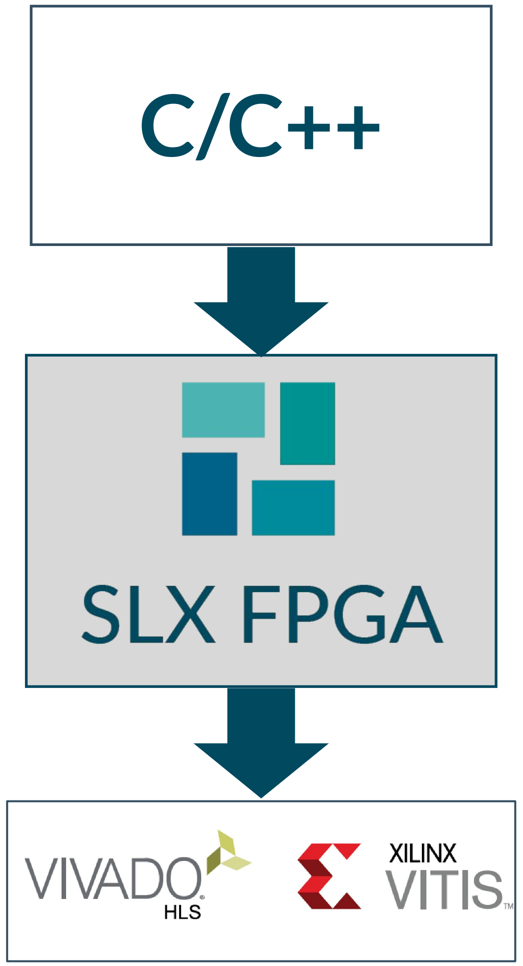
- However, there are challenges…

High-Level Synthesis – First Impressions

- Can’t compile – lots of synthesizability errors
- Slow performance and/or bloated area
- Difficult to detect parallelism and remove parallelism blockers
- Time consuming, iterative manual pragma optimization/insertion

Silexica SLX FPGA

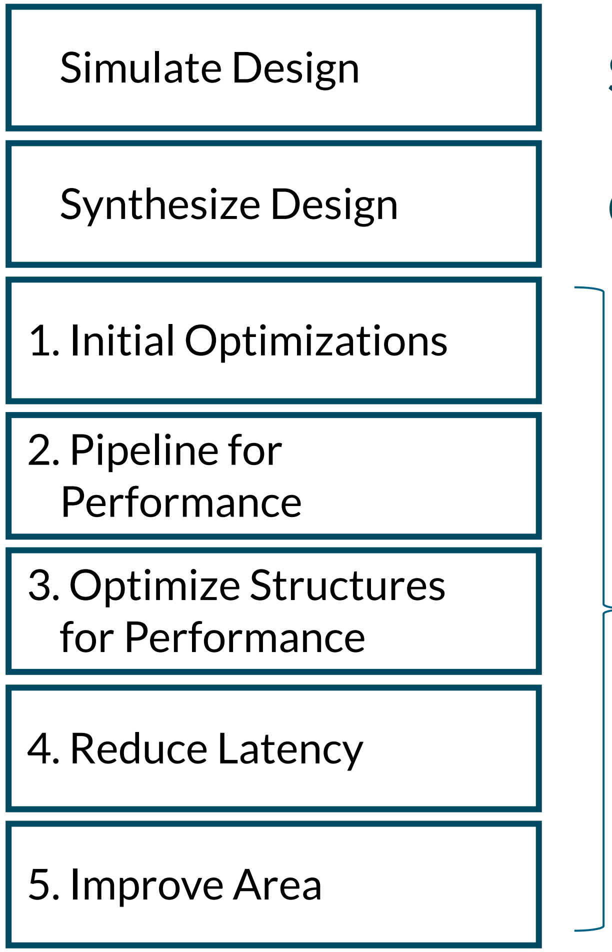
- SLX FPGA sits on top of HLS compiler
  - Prepares the C/C++ code for optimum HLS results
  - Takes the guesswork out of using HLS
- Removes the roadblocks in HLS adoption
  - Non-synthesizable C/C++ code
  - Finding parallelism
  - Poor performance and bloated area
- **HW engineers:** Get SW guidance needed
- **SW engineers:** Get parallelism/HW guidance



Vivado HLS Optimization Methodology Guide

Simulate Design	- Validate the C function
Synthesize Design	- Baseline design
1. Initial Optimizations	- Define interfaces (and data packing) - Define loop trip counts
2. Pipeline for Performance	- Pipeline and dataflow
3. Optimize Structures for Performance	- Partition memories and ports - Remove false dependencies
4. Reduce Latency	- Optionally specify latency requirements
5. Improve Area	- Optionally recover resources through sharing

SLX FPGA Assists at Every Stage

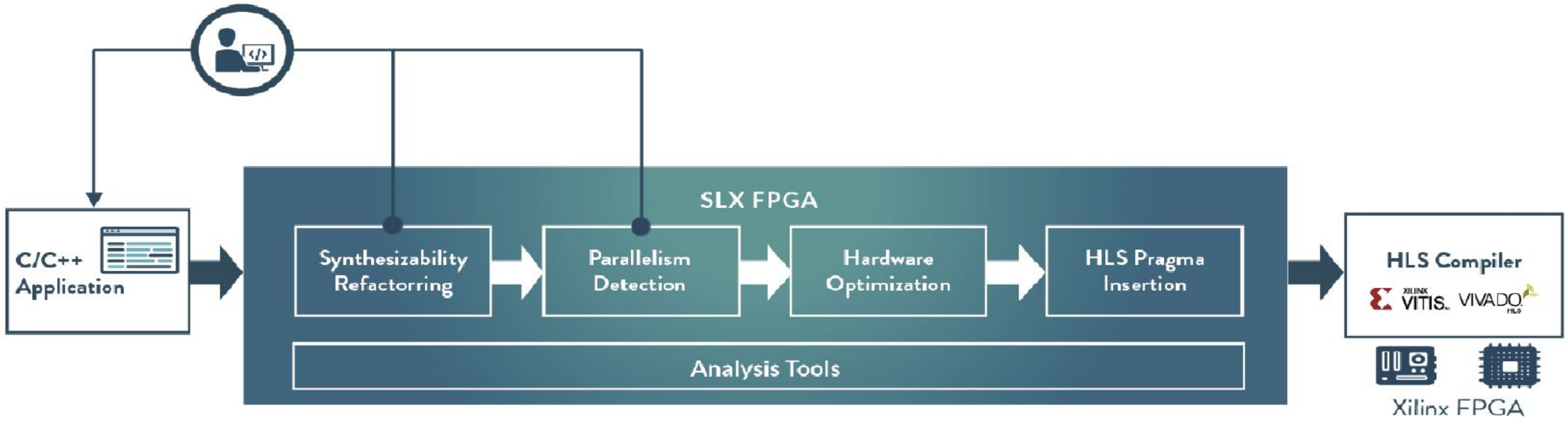


Simulate & debug from SLX

Guided code refactoring

SLX FPGA Capabilities:
- Deep Profiling
- Parallelism Detection
- Intelligent, Constraint-Based Heuristics
- Fast Pragma Exploration
- Analysis Tools

SLX FPGA - Automated Workflow for HLS



Synthesizability Refactoring

Code Refactoring for Synthesizability

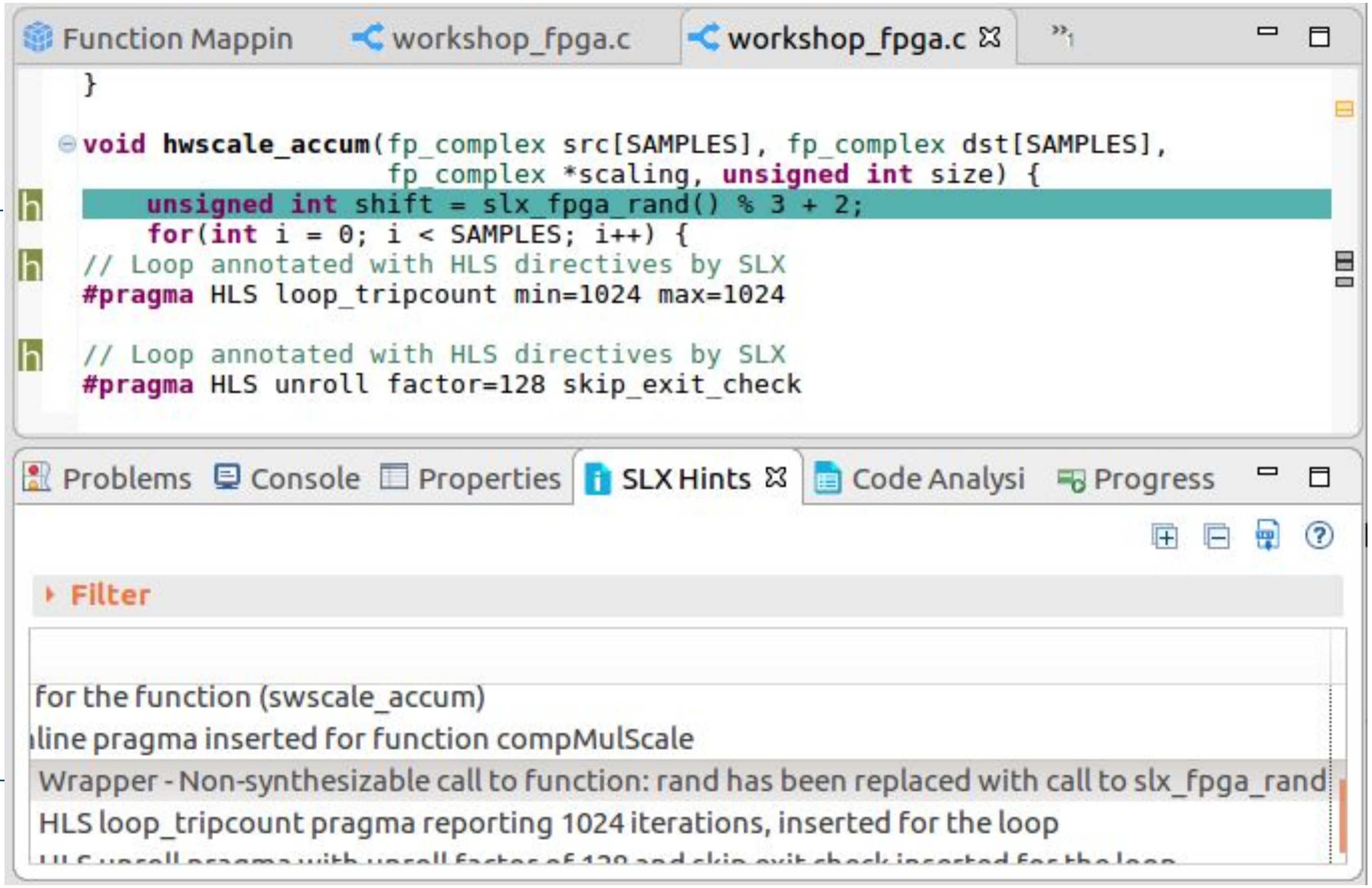
- Not all C/C++ code is compatible with HLS
  - HLS has unique coding standards that must be followed
  - Not trivial – must become fluent in coding C/C++ for HLS

- SLX FPGA helps users refactor code for synthesizability
  - SLX FPGA identifies non-synthesizable functions code
  - Automatically refactors code for some common functions
  - Provides guided refactoring with hints on how to rewrite code

Automatic Code Refactoring

Automatic refactoring of well-known functions

Direct link to generated code



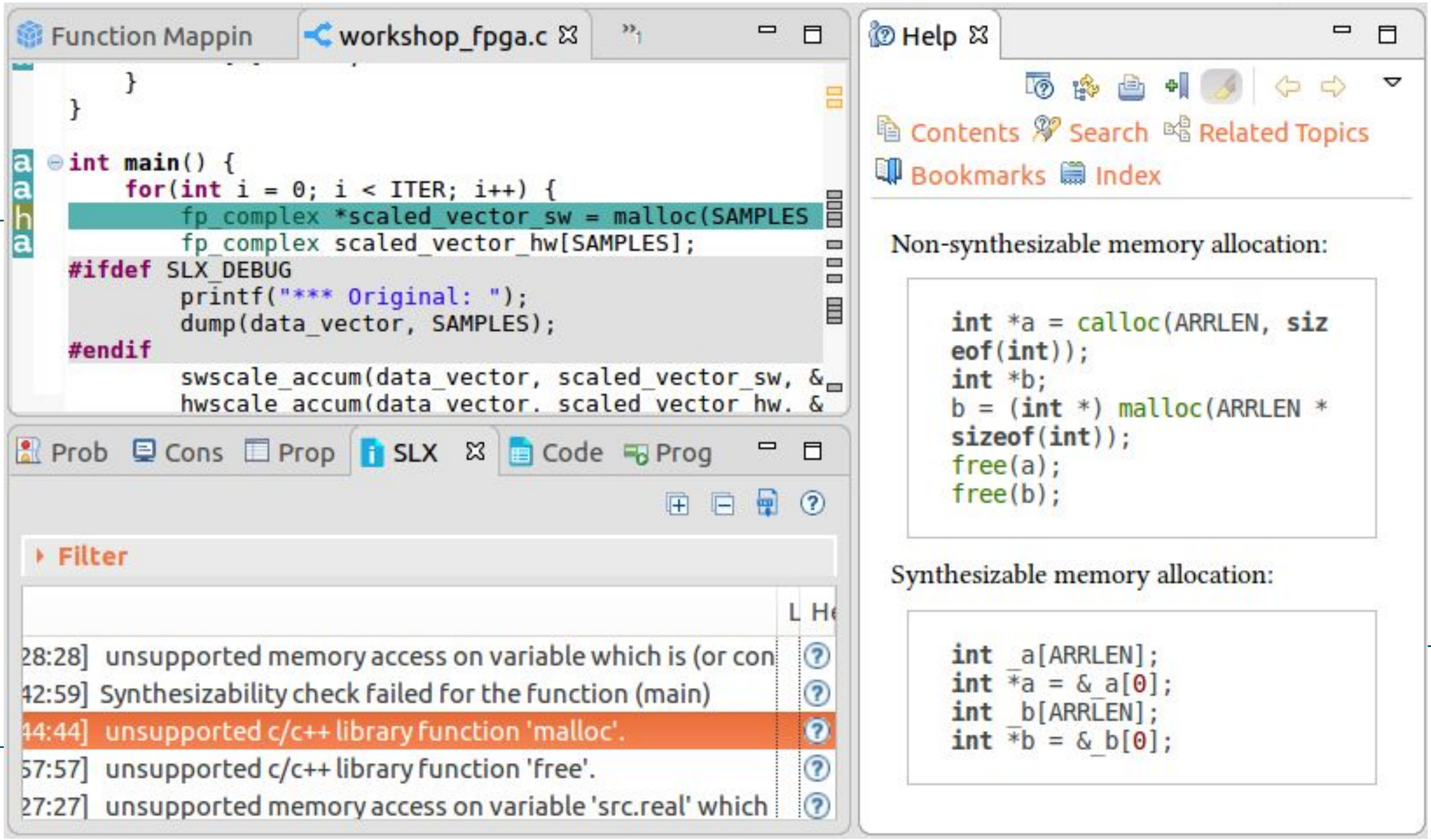
```
}

void hwscale_accum(fp_complex src[SAMPLES], fp_complex dst[SAMPLES],
                   fp_complex *scaling, unsigned int size) {
    unsigned int shift = slx_fpga_rand() % 3 + 2;
    for(int i = 0; i < SAMPLES; i++) {
// Loop annotated with HLS directives by SLX
#pragma HLS loop_tripcount min=1024 max=1024

// Loop annotated with HLS directives by SLX
#pragma HLS unroll factor=128 skip_exit_check
```

```
for the function (swscale_accum)
line pragma inserted for function compMulScale
Wrapper - Non-synthesizable call to function: rand has been replaced with call to slx_fpga_rand
HLS loop_tripcount pragma reporting 1024 iterations, inserted for the loop
```

Guided Code Refactoring

Highlighting of non-synthesizable code (access to pointer)

Summarized report to help refactoring



```
        }
    }

int main() {
    for(int i = 0; i < ITER; i++) {
        fp_complex *scaled_vector_sw = malloc(SAMPLES
        fp_complex scaled_vector_hw[SAMPLES];
#ifdef SLX_DEBUG
        printf("*** Original: ");
        dump(data_vector, SAMPLES);
#endif
        swscale_accum(data_vector, scaled_vector_sw, &
        hwscale_accum(data_vector, scaled_vector_hw, &
```

```
28:28]  unsupported memory access on variable which is (or con
42:59]  Synthesizability check failed for the function (main)
44:44]  unsupported c/c++ library function 'malloc'.
57:57]  unsupported c/c++ library function 'free'.
27:27]  unsupported memory access on variable 'src.real' which
```

Non-synthesizable memory allocation:

```
int *a = calloc(ARRLEN, siz
eof(int));
int *b;
b = (int *) malloc(ARRLEN *
sizeof(int));
free(a);
free(b);
```

Synthesizable memory allocation:

```
int _a[ARRLEN];
int *a = &_a[0];
int _b[ARRLEN];
int *b = &_b[0];
```

Guidance for code re-writing

Parallelism Detection

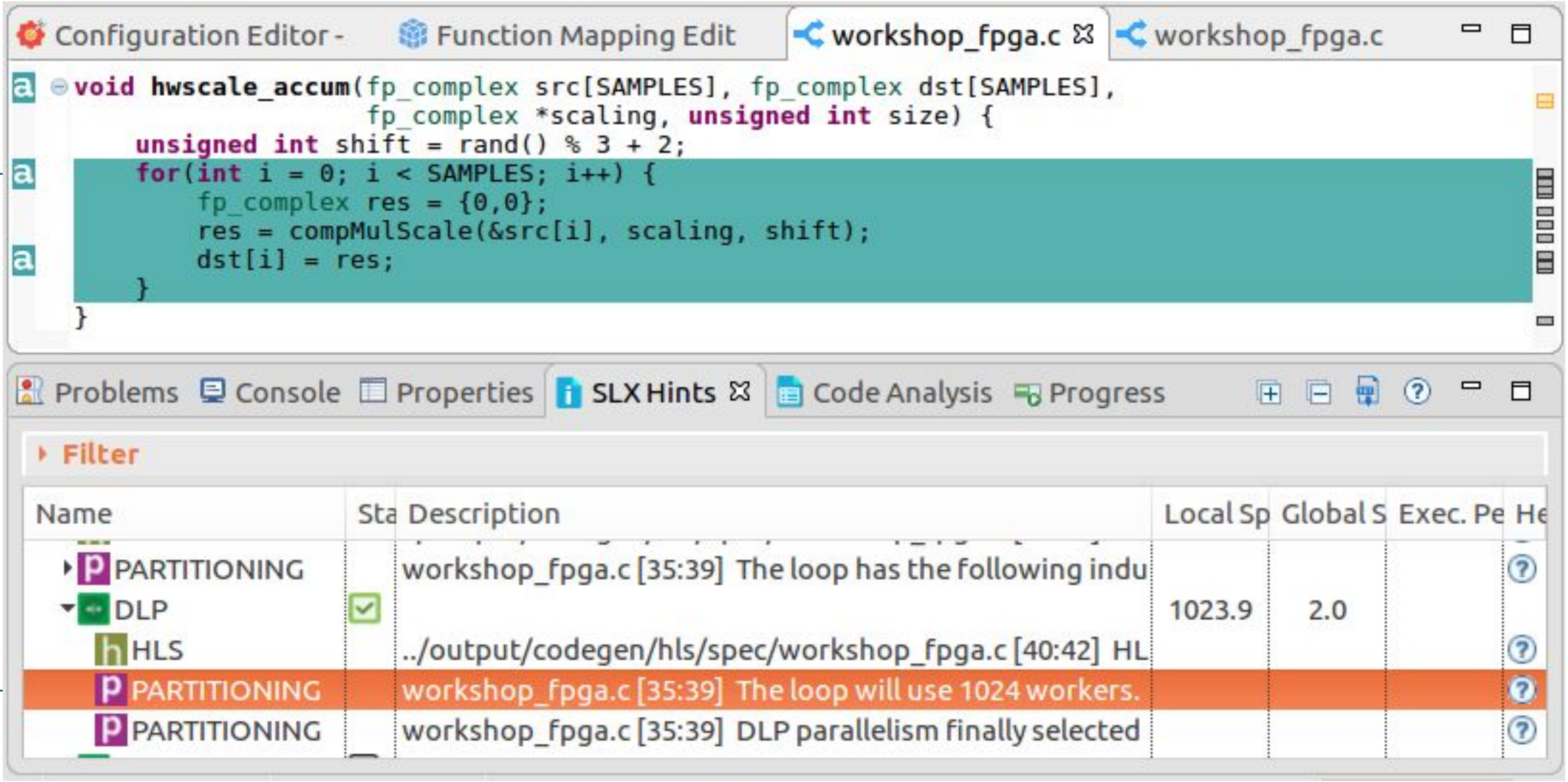
Parallelism Detection

- Identifying parallelism is difficult
  - Even more difficult if HLS user did not write algorithm
- SLX FPGA analyzes the applications and identifies parallelism patterns to implement in hardware
  - Identifies Data Level and Pipeline Level Parallelism
  - Also provides insights into parallelism blockers

Parallelism Detection

Source highlighting for parallel code

Summarized report to help navigation



```
void hwscale_accum(fp_complex src[SAMPLES], fp_complex dst[SAMPLES],
                   fp_complex *scaling, unsigned int size) {
    unsigned int shift = rand() % 3 + 2;
    for(int i = 0; i < SAMPLES; i++) {
        fp_complex res = {0,0};
        res = compMulScale(&src[i], scaling, shift);
        dst[i] = res;
    }
}
```

Name	Sta	Description	Local Sp	Global S	Exec. Pe	He
PARTITIONING		workshop_fpga.c [35:39] The loop has the following indu				?
DLP	✓		1023.9	2.0		
HLS		../output/codegen/hls/spec/workshop_fpga.c [40:42] HL				?
PARTITIONING		workshop_fpga.c [35:39] The loop will use 1024 workers.				?
PARTITIONING		workshop_fpga.c [35:39] DLP parallelism finally selected				?

Parallelism Detection

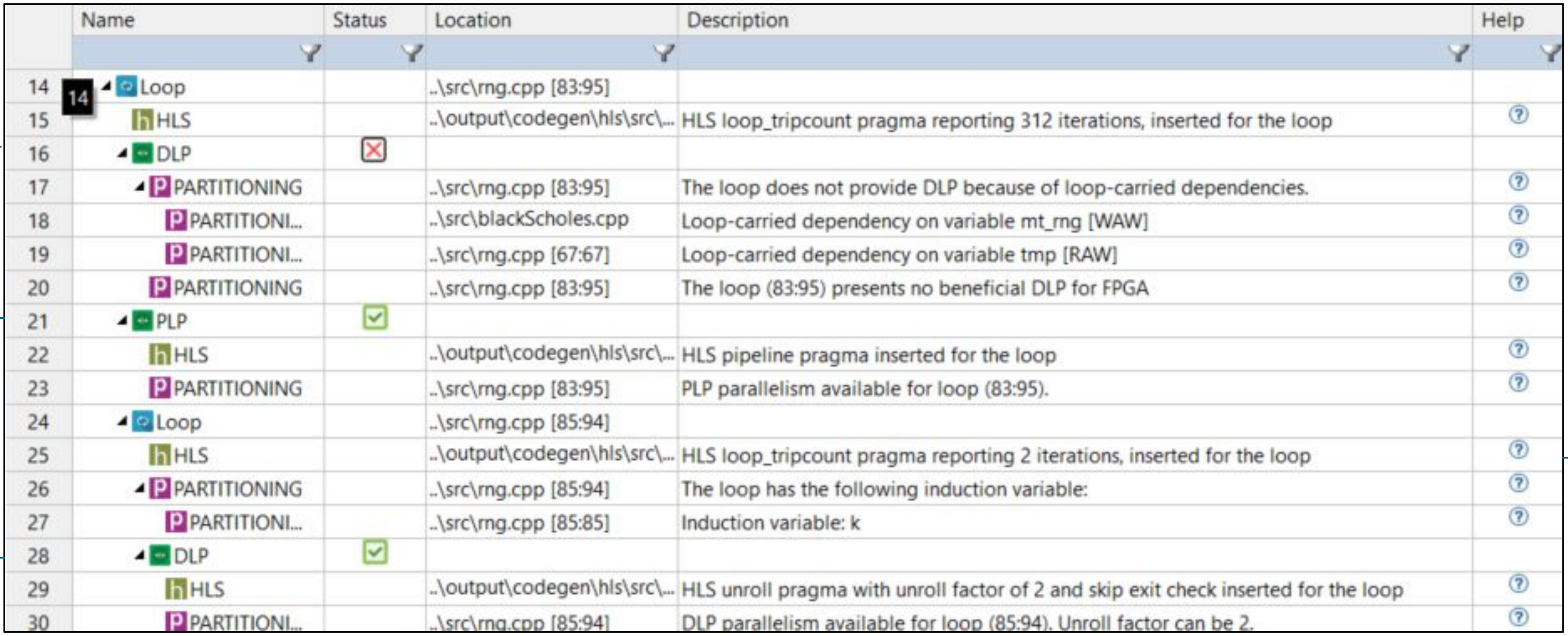
Blocked DLP

PLP

DLP

Blockers identified

Automatic tripcount insertion



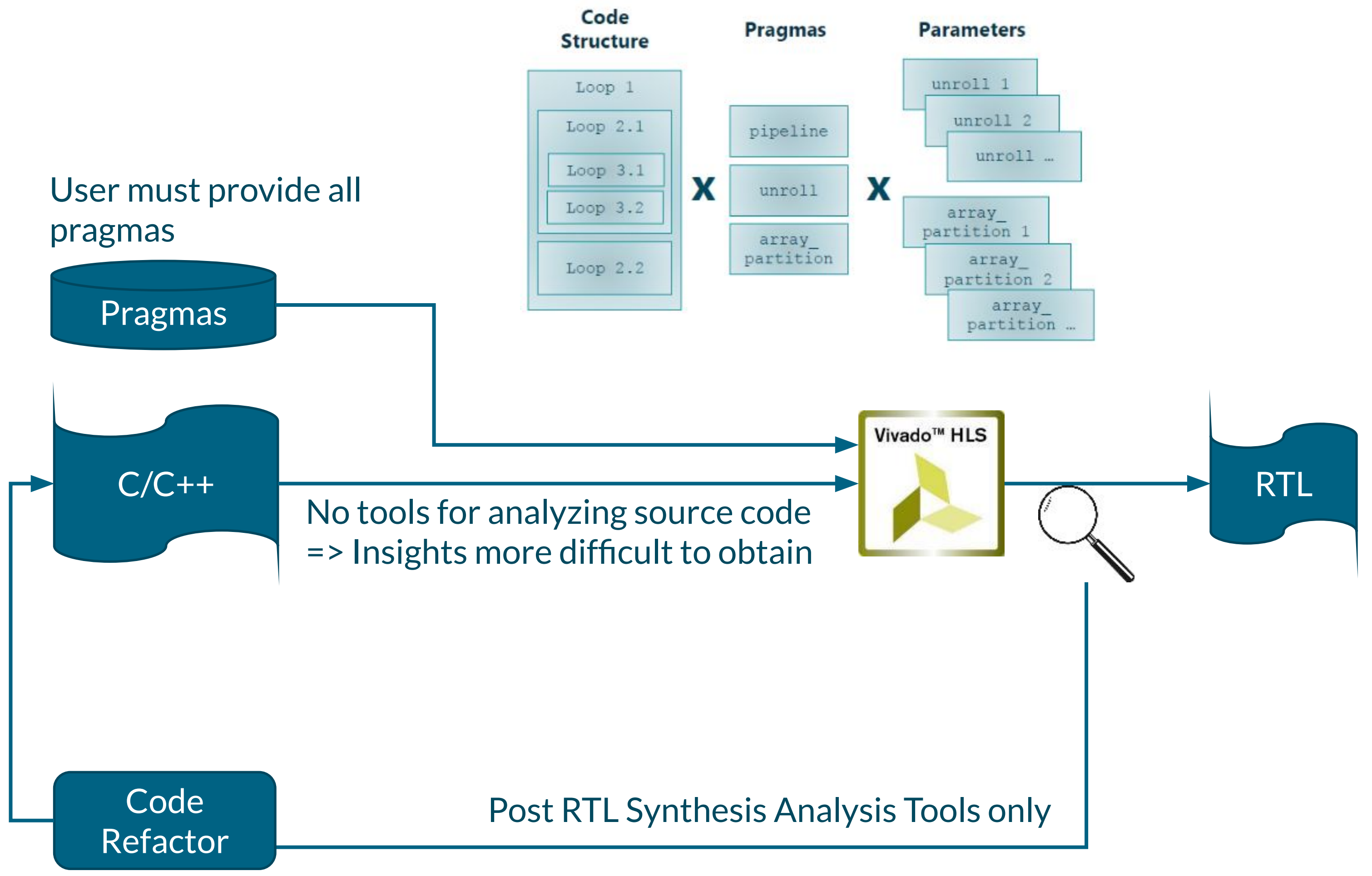
	Name	Status	Location	Description	Help
14	Loop		..\src\rng.cpp [83:95]		
15	HLS		..\output\codegen\hls\src\...	HLS loop_tripcount pragma reporting 312 iterations, inserted for the loop	?
16	DLP	✗			
17	PARTITIONING		..\src\rng.cpp [83:95]	The loop does not provide DLP because of loop-carried dependencies.	?
18	PARTITIONI...		..\src\blackScholes.cpp	Loop-carried dependency on variable mt_rng [WAW]	?
19	PARTITIONI...		..\src\rng.cpp [67:67]	Loop-carried dependency on variable tmp [RAW]	?
20	PARTITIONING		..\src\rng.cpp [83:95]	The loop (83:95) presents no beneficial DLP for FPGA	?
21	PLP	✓			
22	HLS		..\output\codegen\hls\src\...	HLS pipeline pragma inserted for the loop	?
23	PARTITIONING		..\src\rng.cpp [83:95]	PLP parallelism available for loop (83:95).	?
24	Loop		..\src\rng.cpp [85:94]		
25	HLS		..\output\codegen\hls\src\...	HLS loop_tripcount pragma reporting 2 iterations, inserted for the loop	?
26	PARTITIONING		..\src\rng.cpp [85:94]	The loop has the following induction variable:	?
27	PARTITIONI...		..\src\rng.cpp [85:85]	Induction variable: k	?
28	DLP	✓			
29	HLS		..\output\codegen\hls\src\...	HLS unroll pragma with unroll factor of 2 and skip exit check inserted for the loop	?
30	PARTITIONI...		..\src\rng.cpp [85:94]	DLP parallelism available for loop (85:94). Unroll factor can be 2.	?

HW Optimization

Standard Vivado HLS Flow

Pragma Exploration is Tricky

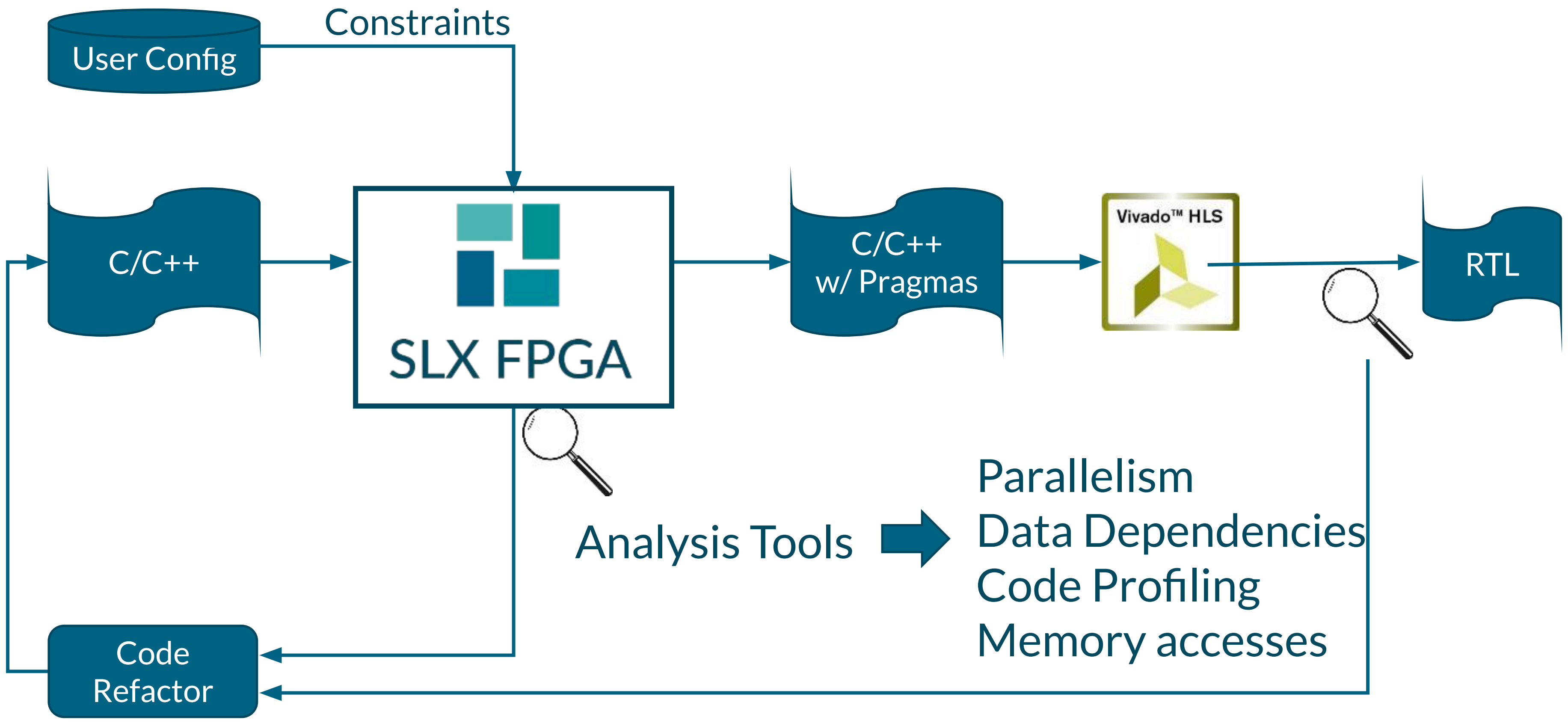
- Powerful, but requires detailed knowledge of code and Vivado HLS
- Even small pragma/parameter set leads to large design space
- Each combination needs to be synthesized with HLS
- Some combinations can lead to bloated implementations, extended synth times



Vivado/Vitis HLS + SLX FPGA Flow

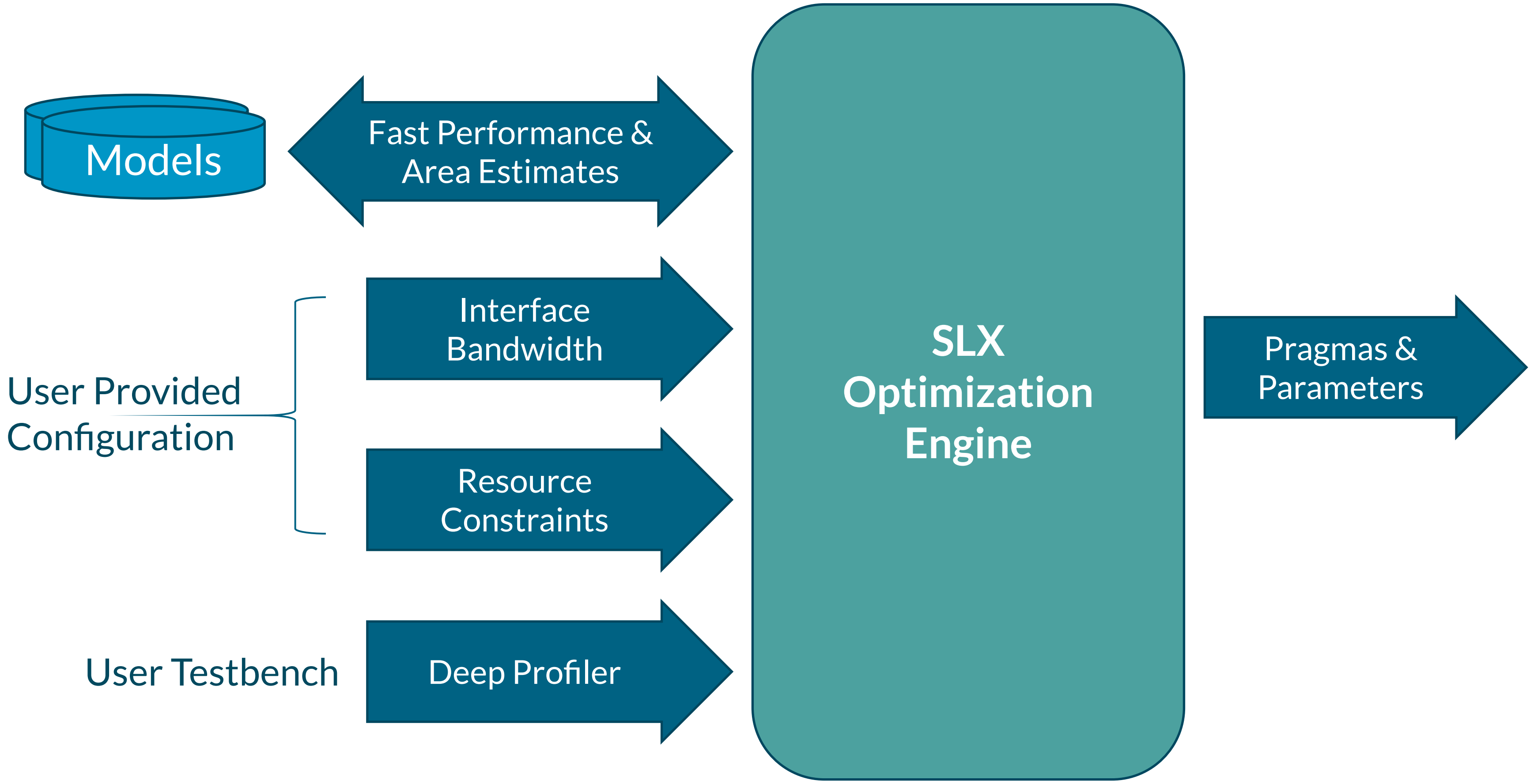
SLX FPGA

- Analyzes and optimizes design based on constraints
- Automatically inserts optimal pragmas into source code
- Analysis tools assist with code refactoring



The SLX Optimization Engine

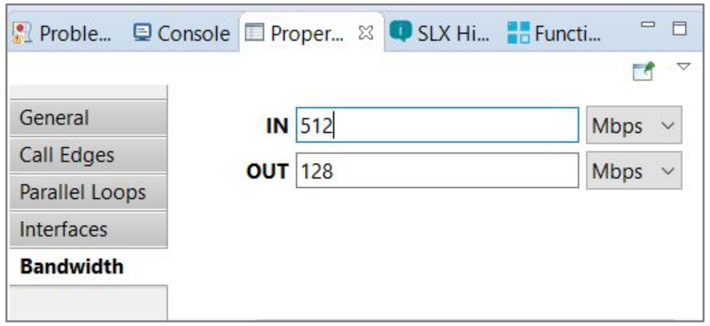
- Fully automated heuristics guided by deep profiling information
- Internal models enable fast architectural exploration
- Ability to explore impacts of different configuration



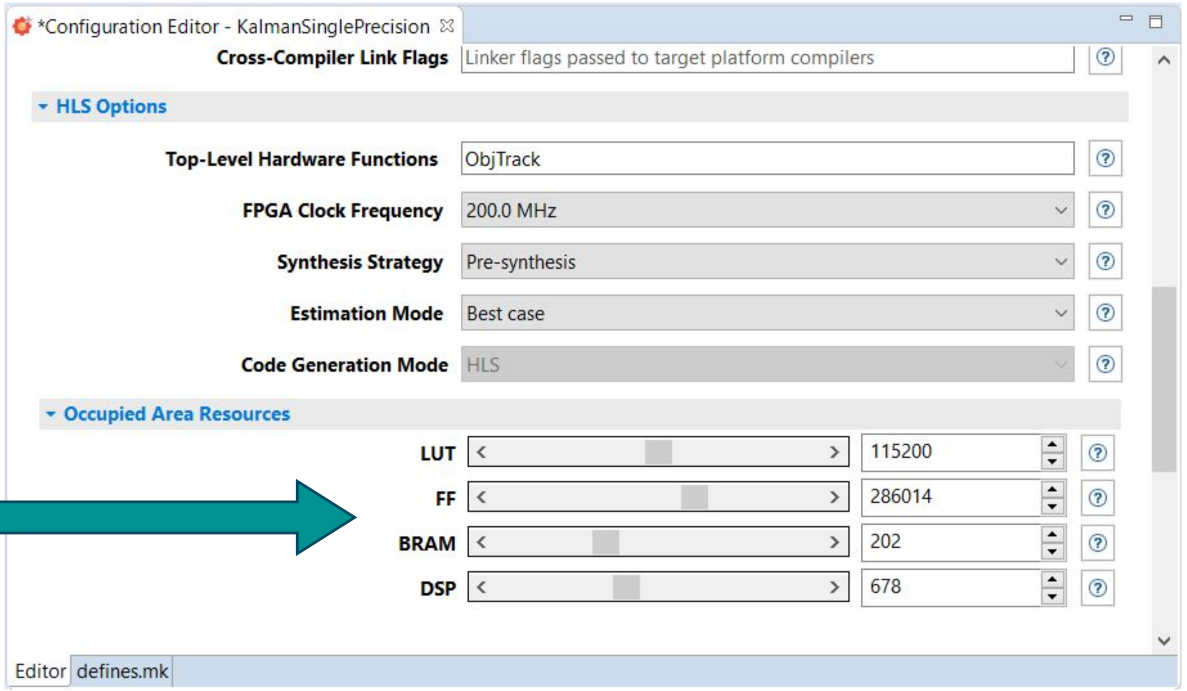
Constraints

Interface bandwidth

Resource constraints



IN	512	Mbps
OUT	128	Mbps


*Configuration Editor - KalmanSinglePrecision

Cross-Compiler Link Flags: Linker flags passed to target platform compilers

HLS Options

Top-Level Hardware Functions	ObjTrack
FPGA Clock Frequency	200.0 MHz
Synthesis Strategy	Pre-synthesis
Estimation Mode	Best case
Code Generation Mode	HLS

Occupied Area Resources

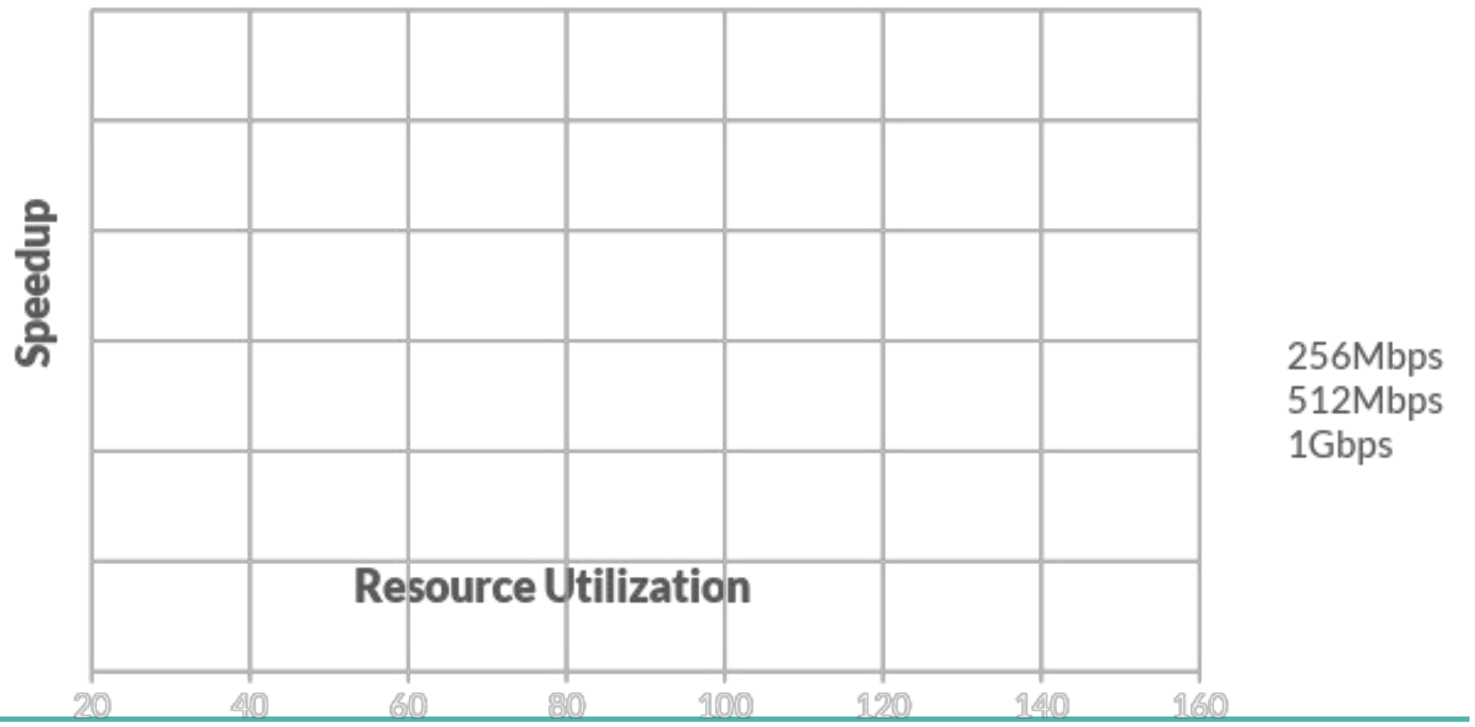
LUT	115200
FF	286014
BRAM	202
DSP	678

SILEXICA

Design Space Exploration (DSE)

- Evaluate multiple user configuration parameters
- Enables area/performance trade-off analysis

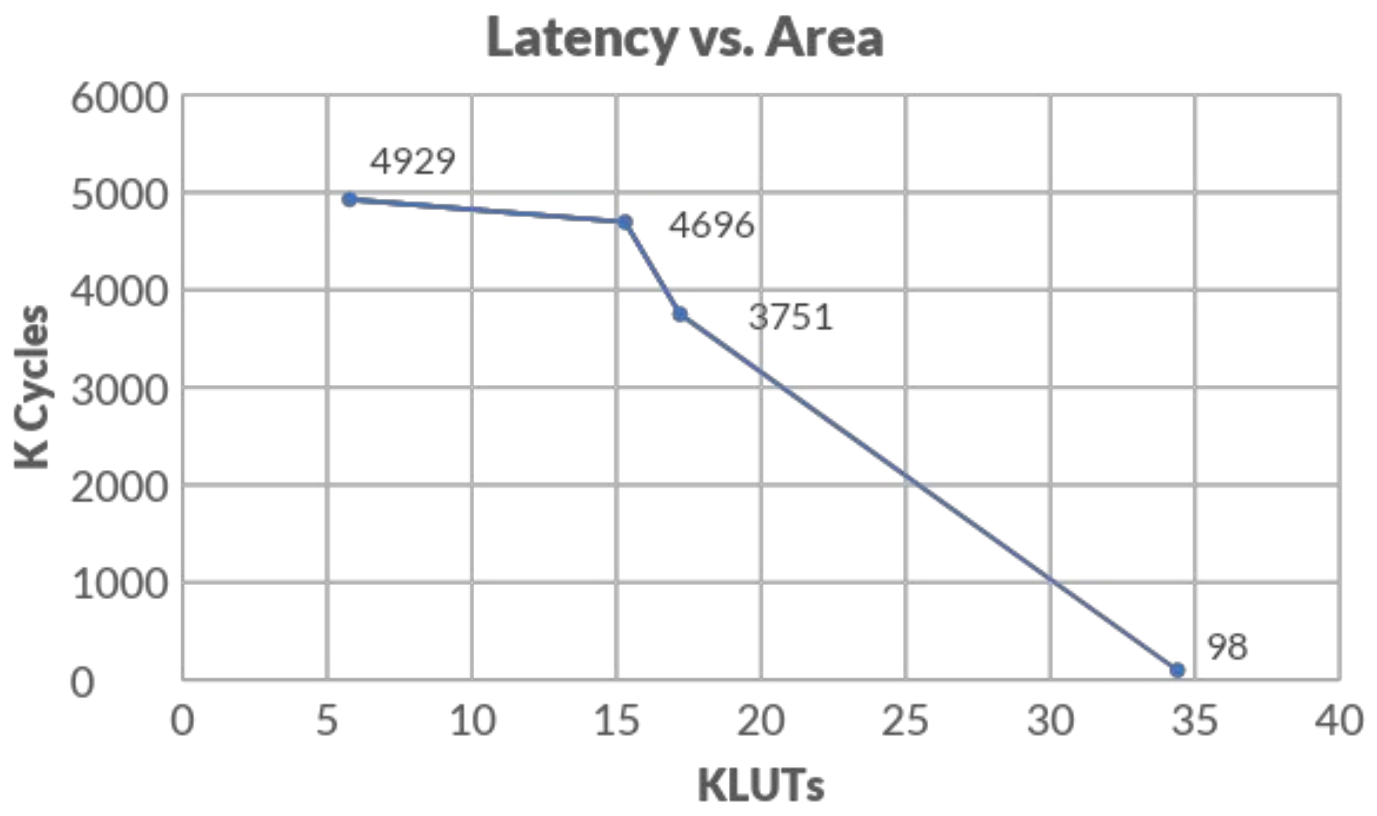
Example:
Use SLX to generate results for
3 interface bandwidth constraints across 8 resource constraints



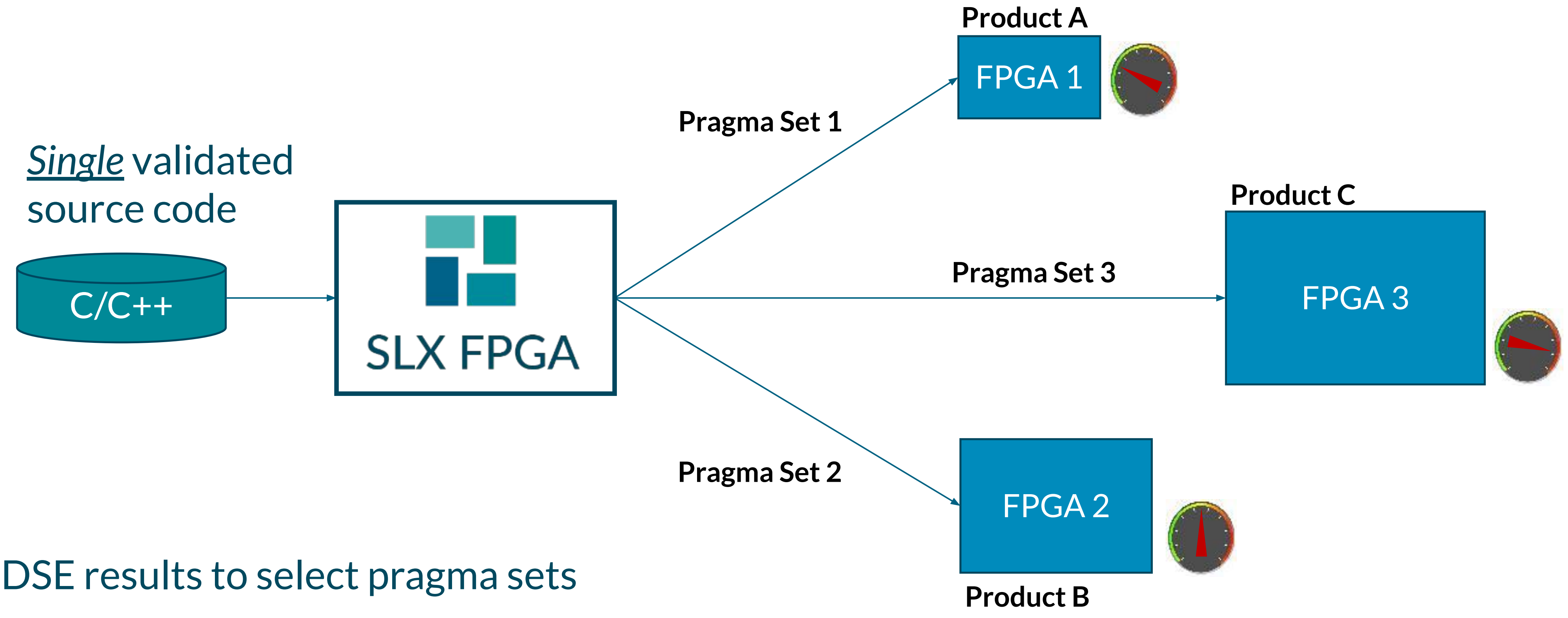
SILEXICA

Kalman Filter Example

SLX Constraint (KLUTs)	Actual LUT usage from HLS (KLUTs)	Latency (1000s of clock cycles)
Tripcount only	5.8	4929
12	15.3	4696
24	17.2	3751
48	34.4	98
Unlimited	34.4	98



IP Re-use



Use DSE results to select pragma sets

Pragma Insertion

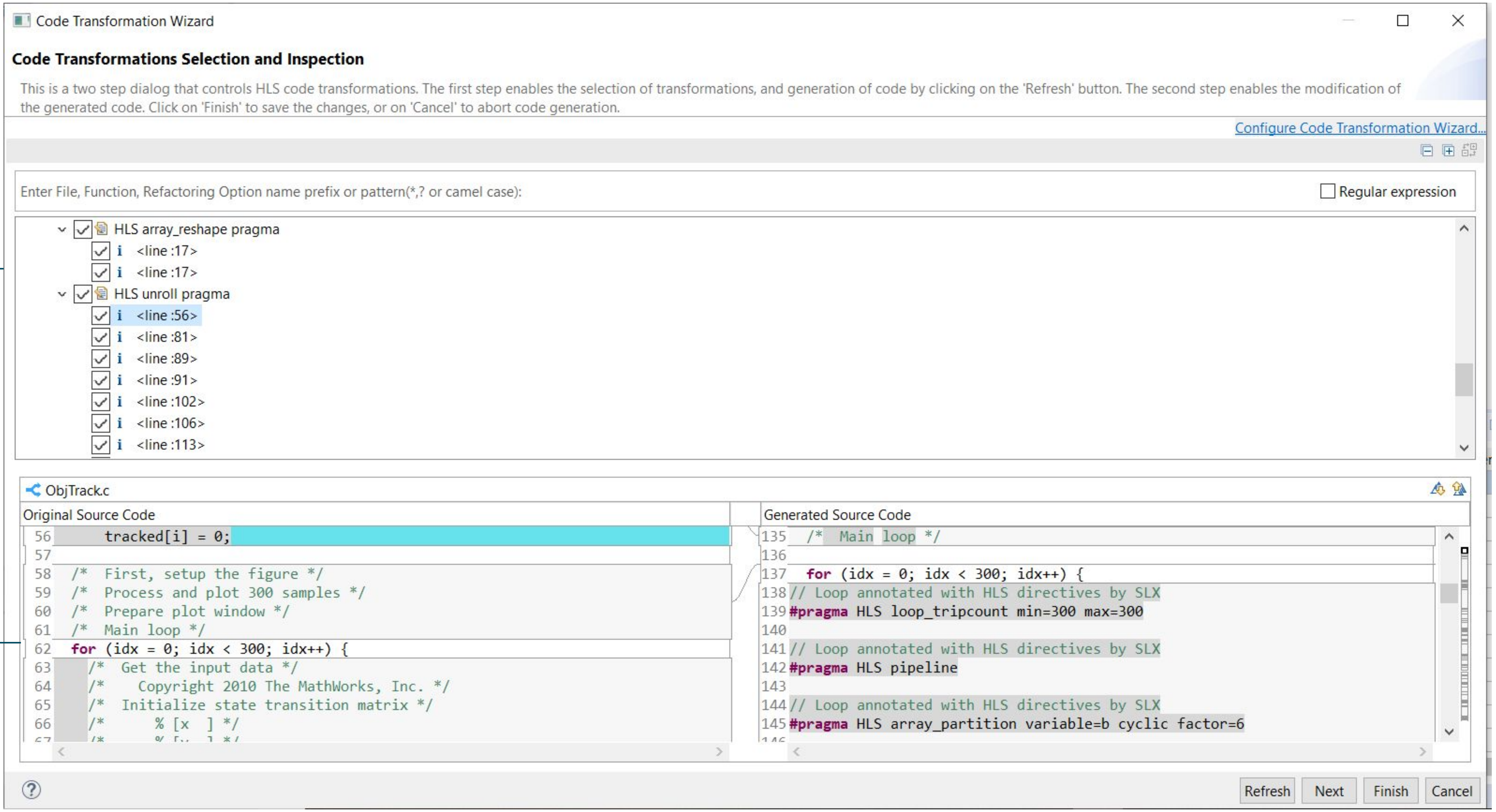
Automatic Pragma Insertion

- HW optimization stage
  - Creates optimal pragma set
- Pragma Insertion Wizard
  - Inserts all generated pragmas into original source code
  - Allows designer full control over pragma insertion

SLX Code Transformation Wizard

Enable/disable individual or groups of pragmas

Original/Updated code side-by-side view



Code Transformations Selection and Inspection

This is a two step dialog that controls HLS code transformations. The first step enables the selection of transformations, and generation of code by clicking on the 'Refresh' button. The second step enables the modification of the generated code. Click on 'Finish' to save the changes, or on 'Cancel' to abort code generation.

Configure Code Transformation Wizard...
Enter File, Function, Refactoring Option name prefix or pattern(*,? or camel case):
Regular expression

```
[✓] HLS array_reshape pragma
    [✓] i <line :17>
    [✓] i <line :17>
[✓] HLS unroll pragma
    [✓] i <line :56>
    [✓] i <line :81>
    [✓] i <line :89>
    [✓] i <line :91>
    [✓] i <line :102>
    [✓] i <line :106>
    [✓] i <line :113>
```

Original Source Code

```
56        tracked[i] = 0;
57
58   /*  First, setup the figure */
59   /*  Process and plot 300 samples */
60   /*  Prepare plot window */
61   /*  Main loop */
62   for (idx = 0; idx < 300; idx++) {
63     /*  Get the input data */
64     /*    Copyright 2010 The MathWorks, Inc. */
65     /*  Initialize state transition matrix */
66     /*      % [x  ] */
```

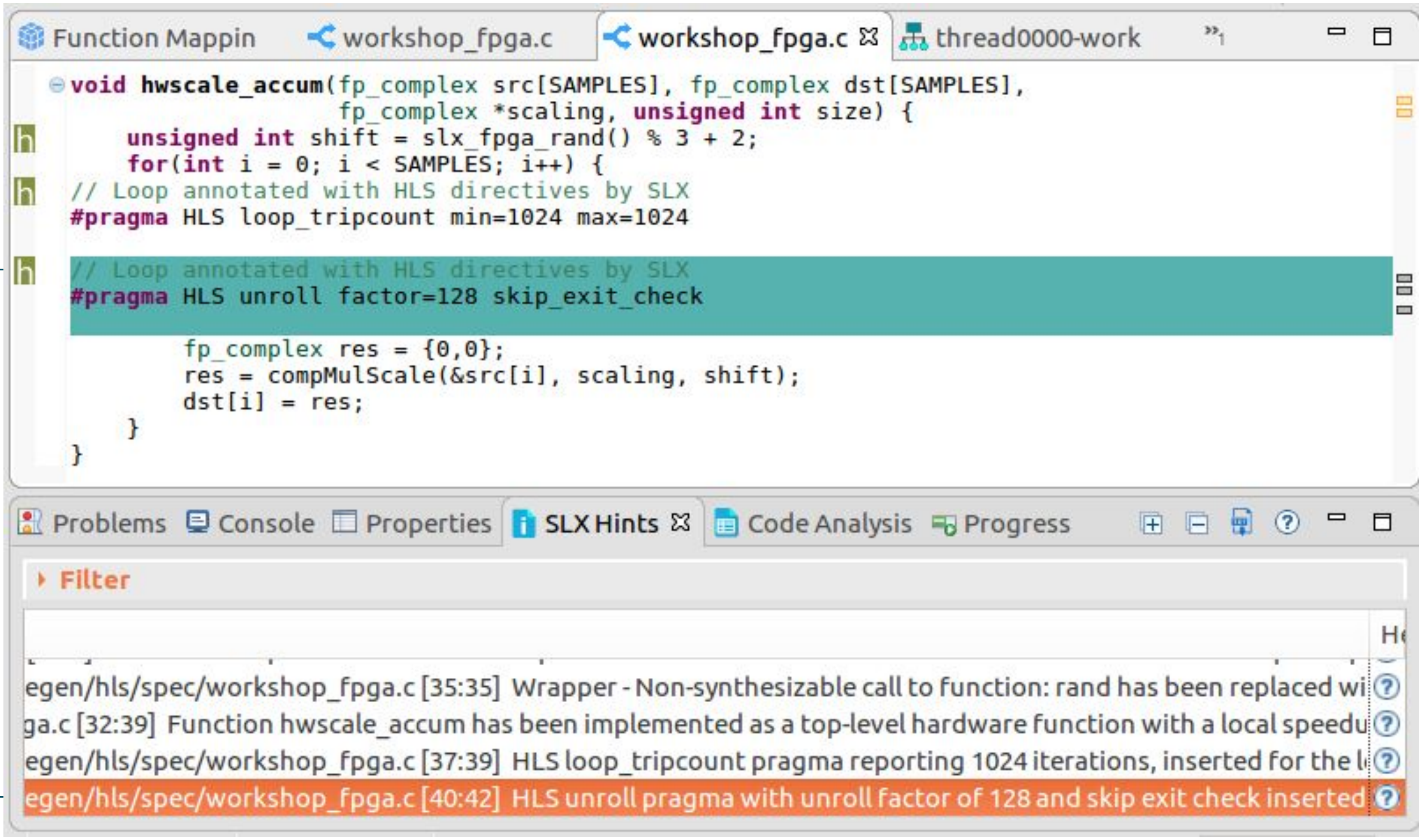
Generated Source Code

```
135   /*  Main loop */
136
137   for (idx = 0; idx < 300; idx++) {
138 // Loop annotated with HLS directives by SLX
139 #pragma HLS loop_tripcount min=300 max=300
140
141 // Loop annotated with HLS directives by SLX
142 #pragma HLS pipeline
143
144 // Loop annotated with HLS directives by SLX
145 #pragma HLS array_partition variable=b cyclic factor=6
```

Automatic HLS Pragma Insertion

Automatically annotate the code with pragmas

Direct link to generated file



```
void hwscale_accum(fp_complex src[SAMPLES], fp_complex dst[SAMPLES],
                   fp_complex *scaling, unsigned int size) {
    unsigned int shift = slx_fpga_rand() % 3 + 2;
    for(int i = 0; i < SAMPLES; i++) {
// Loop annotated with HLS directives by SLX
#pragma HLS loop_tripcount min=1024 max=1024

// Loop annotated with HLS directives by SLX
#pragma HLS unroll factor=128 skip_exit_check

        fp_complex res = {0,0};
        res = compMulScale(&src[i], scaling, shift);
        dst[i] = res;
    }
}
```

```
egen/hls/spec/workshop_fpga.c [35:35]  Wrapper - Non-synthesizable call to function: rand has been replaced wi
ga.c [32:39]  Function hwscale_accum has been implemented as a top-level hardware function with a local speedu
egen/hls/spec/workshop_fpga.c [37:39]  HLS loop_tripcount pragma reporting 1024 iterations, inserted for the l
egen/hls/spec/workshop_fpga.c [40:42]  HLS unroll pragma with unroll factor of 128 and skip exit check inserted
```

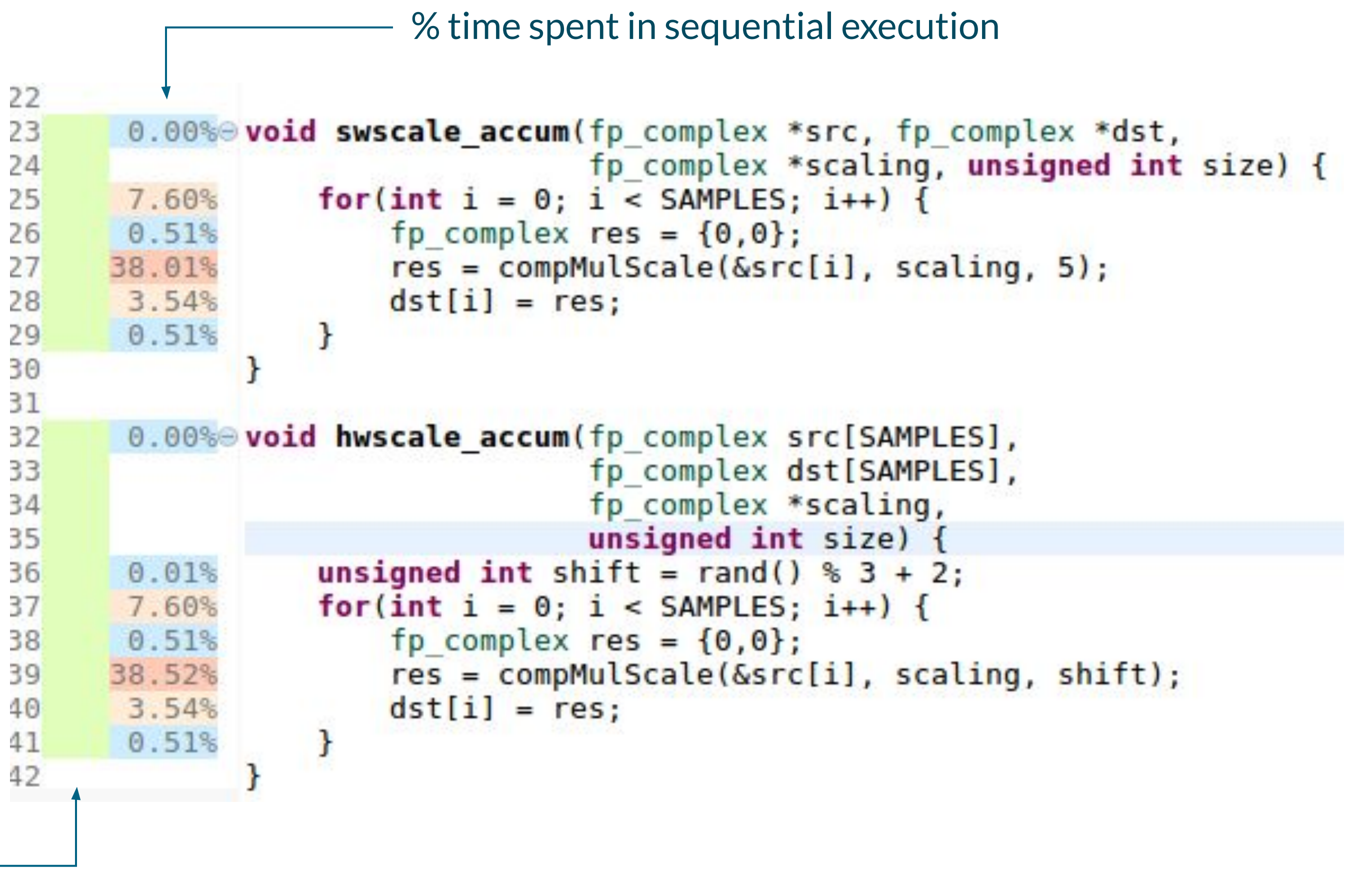
SILEXICA

Deep Profiling and Analysis Tools

Hardware Aware Coding

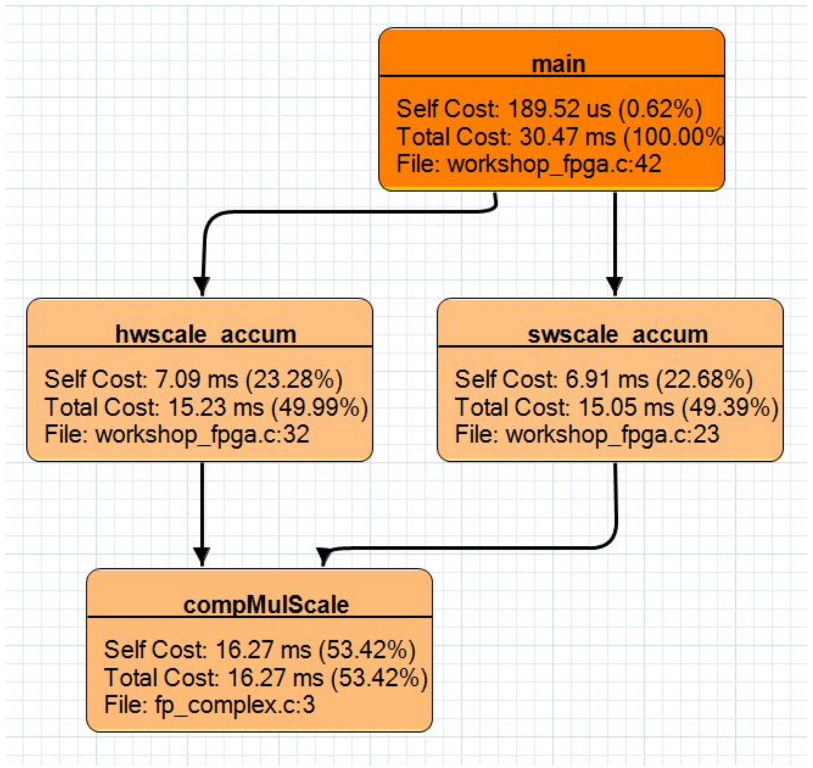
- Pragmas can only go so far
    - Modifying the original code to be more hardware aware can lead to better performance and area
- Hardware aware refactoring requires insights into the algorithm.
- SLX Analysis tools provide detailed insights on the source code:
    - Code and Function Profiling
    - DLP and PLP detection
    - Data dependency detection
    - Software Call Graphs
    - Hotspot detection
    - Memory/variable analysis
    - Code analysis graphs

Code Profiling

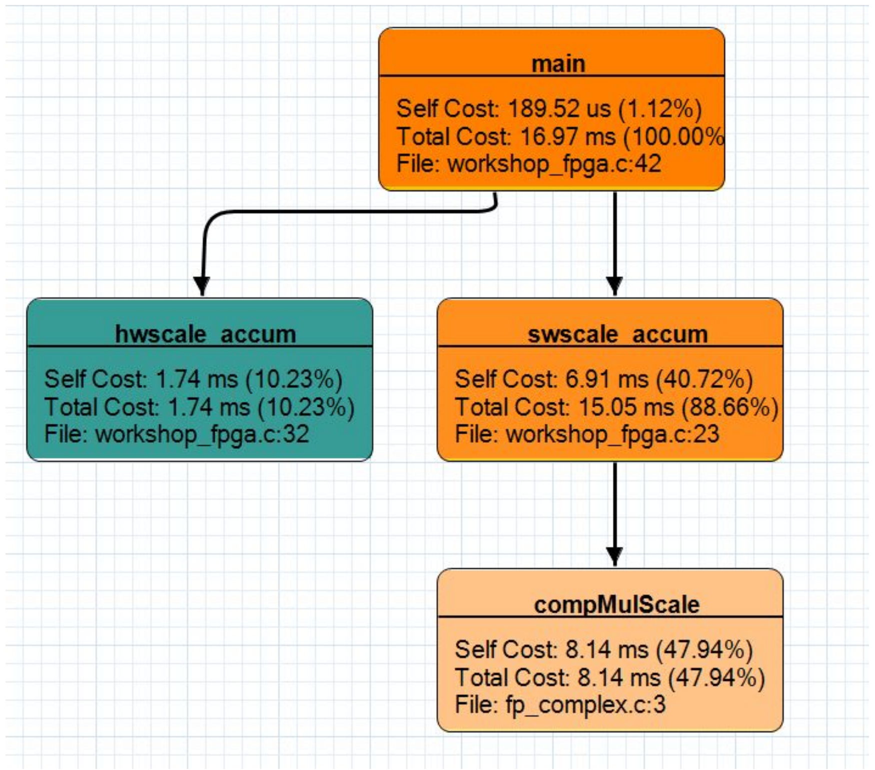


```
0.00% void swscale_accum(fp_complex *src, fp_complex *dst,
                         fp_complex *scaling, unsigned int size) {
7.60%     for(int i = 0; i < SAMPLES; i++) {
0.51%         fp_complex res = {0,0};
38.01%        res = compMulScale(&src[i], scaling, 5);
3.54%         dst[i] = res;
0.51%     }
      }

0.00% void hwscale_accum(fp_complex src[SAMPLES],
                         fp_complex dst[SAMPLES],
                         fp_complex *scaling,
                         unsigned int size) {
0.01%     unsigned int shift = rand() % 3 + 2;
7.60%     for(int i = 0; i < SAMPLES; i++) {
0.51%         fp_complex res = {0,0};
38.52%        res = compMulScale(&src[i], scaling, shift);
3.54%         dst[i] = res;
0.51%     }
      }
```

Software Call Graphs with Profiling

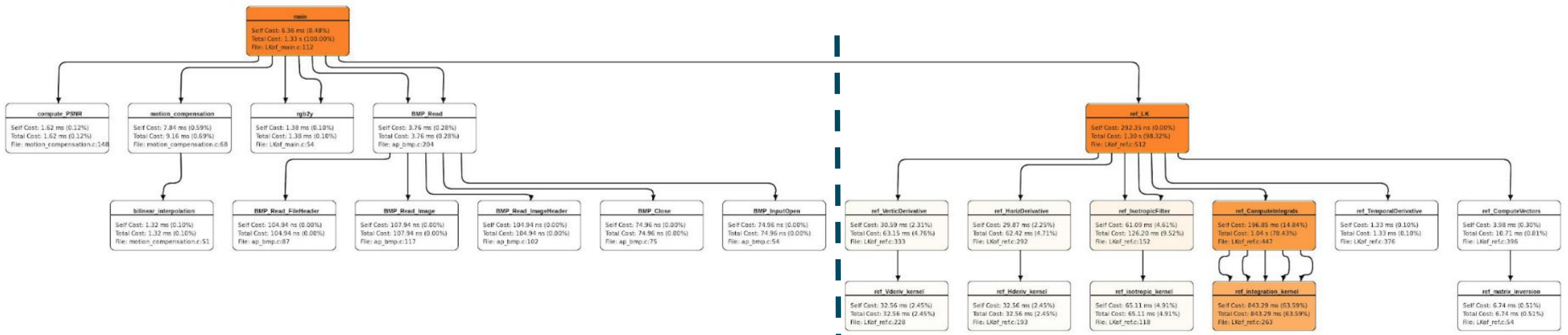

main
Self Cost: 189.52 us (0.62%)
Total Cost: 30.47 ms (100.00%
File: workshop_fpga.c:42
hwscale_accum
Self Cost: 7.09 ms (23.28%)
Total Cost: 15.23 ms (49.99%)
File: workshop_fpga.c:32
swscale_accum
Self Cost: 6.91 ms (22.68%)
Total Cost: 15.05 ms (49.39%)
File: workshop_fpga.c:23
compMulScale
Self Cost: 16.27 ms (53.42%)
Total Cost: 16.27 ms (53.42%)
File: fp_complex.c:3

Pre-Optimization


main
Self Cost: 189.52 us (1.12%)
Total Cost: 16.97 ms (100.00%
File: workshop_fpga.c:42
hwscale_accum
Self Cost: 1.74 ms (10.23%)
Total Cost: 1.74 ms (10.23%)
File: workshop_fpga.c:32
swscale_accum
Self Cost: 6.91 ms (40.72%)
Total Cost: 15.05 ms (88.66%)
File: workshop_fpga.c:23
compMulScale
Self Cost: 8.14 ms (47.94%)
Total Cost: 8.14 ms (47.94%)
File: fp_complex.c:3

Post-Optimization

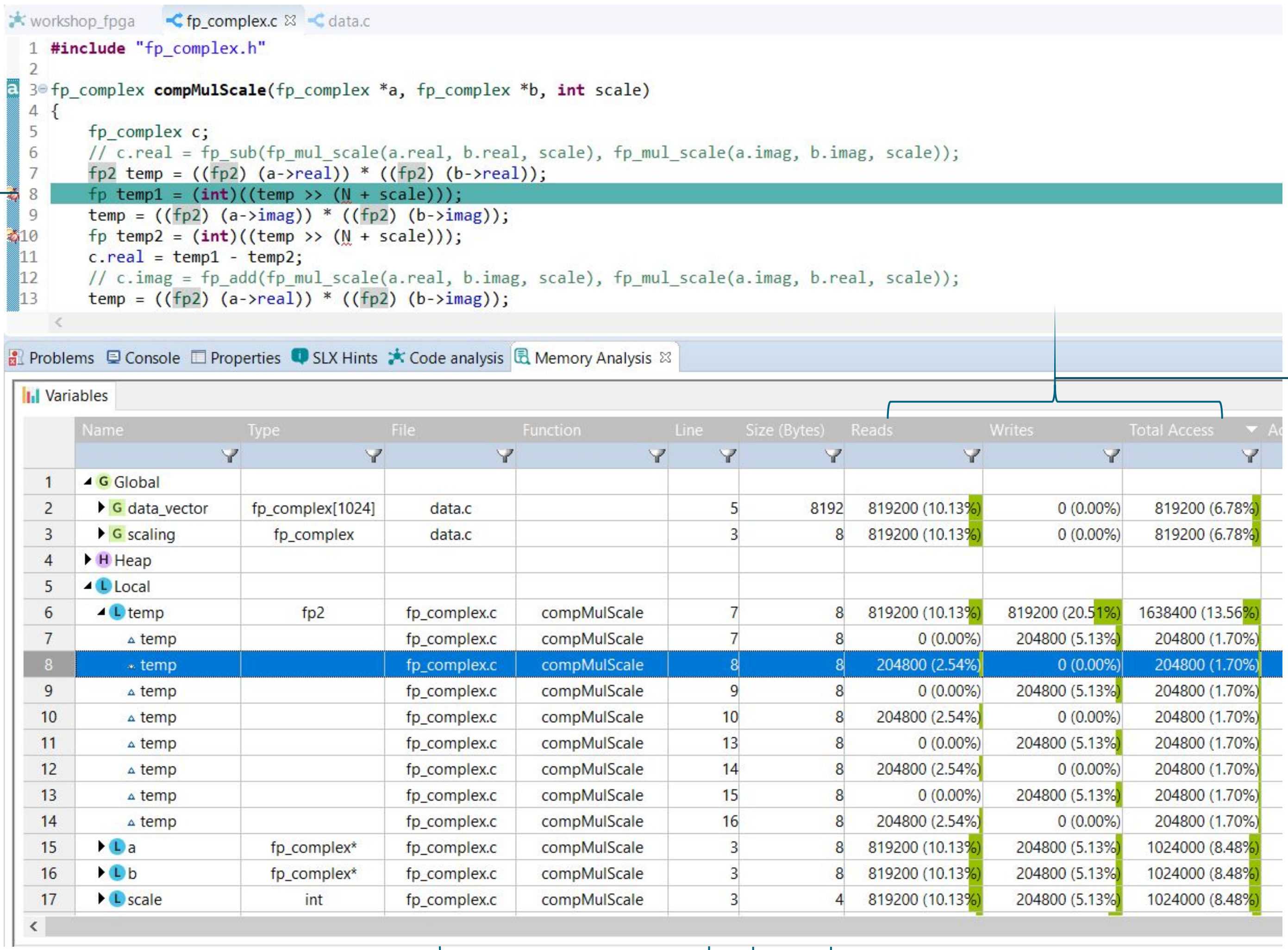
Hotspot Detection


main
Self Cost: 6.36 ms (0.48%)
Total Cost: 1.33 s (100.00%)
File: LKof_main.c:112
compute_PSNR
Self Cost: 1.62 ms (0.12%)
Total Cost: 1.62 ms (0.12%)
File: motion_compensation.c:148
motion_compensation
Self Cost: 7.84 ms (0.59%)
Total Cost: 9.16 ms (0.69%)
File: motion_compensation.c:68
rgb2y
Self Cost: 1.38 ms (0.10%)
Total Cost: 1.38 ms (0.10%)
File: LKof_main.c:54
BMP_Read
Self Cost: 3.76 ms (0.28%)
Total Cost: 3.76 ms (0.28%)
File: ap_bmp.c:204
bilinear_interpolation
Self Cost: 1.32 ms (0.10%)
Total Cost: 1.32 ms (0.10%)
File: motion_compensation.c:51
BMP_Read_FileHeader
Self Cost: 104.94 ns (0.00%)
Total Cost: 104.94 ns (0.00%)
File: ap_bmp.c:87
BMP_Read_Image
Self Cost: 107.94 ns (0.00%)
Total Cost: 107.94 ns (0.00%)
File: ap_bmp.c:117
BMP_Read_ImageHeader
Self Cost: 104.94 ns (0.00%)
Total Cost: 104.94 ns (0.00%)
File: ap_bmp.c:102
BMP_Close
Self Cost: 74.96 ns (0.00%)
Total Cost: 74.96 ns (0.00%)
File: ap_bmp.c:75
BMP_InputOpen
Self Cost: 74.96 ns (0.00%)
Total Cost: 74.96 ns (0.00%)
File: ap_bmp.c:54
ref_LK
Self Cost: 292.35 ns (0.00%)
Total Cost: 1.30 s (98.32%)
File: LKof_ref.c:512
ref_VerticDerivative
Self Cost: 30.59 ms (2.31%)
Total Cost: 63.15 ms (4.76%)
File: LKof_ref.c:333
ref_HorizDerivative
Self Cost: 29.87 ms (2.25%)
Total Cost: 62.42 ms (4.71%)
File: LKof_ref.c:292
ref_IsotropicFilter
Self Cost: 61.09 ms (4.61%)
Total Cost: 126.20 ms (9.52%)
File: LKof_ref.c:152
ref_ComputeIntegrals
Self Cost: 196.85 ms (14.84%)
Total Cost: 1.04 s (78.43%)
File: LKof_ref.c:447
ref_TemporalDerivative
Self Cost: 1.33 ms (0.10%)
Total Cost: 1.33 ms (0.10%)
File: LKof_ref.c:376
ref_ComputeVectors
Self Cost: 3.98 ms (0.30%)
Total Cost: 10.71 ms (0.81%)
File: LKof_ref.c:396
ref_Vderiv_kernel
Self Cost: 32.56 ms (2.45%)
Total Cost: 32.56 ms (2.45%)
File: LKof_ref.c:228
ref_Hderiv_kernel
Self Cost: 32.56 ms (2.45%)
Total Cost: 32.56 ms (2.45%)
File: LKof_ref.c:193
ref_isotropic_kernel
Self Cost: 65.11 ms (4.91%)
Total Cost: 65.11 ms (4.91%)
File: LKof_ref.c:118
ref_integration_kernel
Self Cost: 843.29 ms (63.59%)
Total Cost: 843.29 ms (63.59%)
File: LKof_ref.c:263
ref_matrix_inversion
Self Cost: 6.74 ms (0.51%)
Total Cost: 6.74 ms (0.51%)
File: LKof_ref.c:54

Testbench Code

Synthesizable Code

Memory Analysis



```
1  #include "fp_complex.h"
2
3  fp_complex compMulScale(fp_complex *a, fp_complex *b, int scale)
4  {
5      fp_complex c;
6      // c.real = fp_sub(fp_mul_scale(a.real, b.real, scale), fp_mul_scale(a.imag, b.imag, scale));
7      fp2 temp = ((fp2) (a->real)) * ((fp2) (b->real));
8      fp temp1 = (int)((temp >> (N + scale)));
9      temp = ((fp2) (a->imag)) * ((fp2) (b->imag));
10     fp temp2 = (int)((temp >> (N + scale)));
11     c.real = temp1 - temp2;
12     // c.imag = fp_add(fp_mul_scale(a.real, b.imag, scale), fp_mul_scale(a.imag, b.real, scale));
13     temp = ((fp2) (a->real)) * ((fp2) (b->imag));
```

	Name	Type	File	Function	Line	Size (Bytes)	Reads	Writes	Total Access
1	G Global								
2	G data_vector	fp_complex[1024]	data.c		5	8192	819200 (10.13%)	0 (0.00%)	819200 (6.78%)
3	G scaling	fp_complex	data.c		3	8	819200 (10.13%)	0 (0.00%)	819200 (6.78%)
4	H Heap								
5	L Local								
6	L temp	fp2	fp_complex.c	compMulScale	7	8	819200 (10.13%)	819200 (20.51%)	1638400 (13.56%)
7	temp		fp_complex.c	compMulScale	7	8	0 (0.00%)	204800 (5.13%)	204800 (1.70%)
8	temp		fp_complex.c	compMulScale	8	8	204800 (2.54%)	0 (0.00%)	204800 (1.70%)
9	temp		fp_complex.c	compMulScale	9	8	0 (0.00%)	204800 (5.13%)	204800 (1.70%)
10	temp		fp_complex.c	compMulScale	10	8	204800 (2.54%)	0 (0.00%)	204800 (1.70%)
11	temp		fp_complex.c	compMulScale	13	8	0 (0.00%)	204800 (5.13%)	204800 (1.70%)
12	temp		fp_complex.c	compMulScale	14	8	204800 (2.54%)	0 (0.00%)	204800 (1.70%)
13	temp		fp_complex.c	compMulScale	15	8	0 (0.00%)	204800 (5.13%)	204800 (1.70%)
14	temp		fp_complex.c	compMulScale	16	8	204800 (2.54%)	0 (0.00%)	204800 (1.70%)
15	L a	fp_complex*	fp_complex.c	compMulScale	3	8	819200 (10.13%)	204800 (5.13%)	1024000 (8.48%)
16	L b	fp_complex*	fp_complex.c	compMulScale	3	8	819200 (10.13%)	204800 (5.13%)	1024000 (8.48%)
17	L scale	int	fp_complex.c	compMulScale	3	4	819200 (10.13%)	204800 (5.13%)	1024000 (8.48%)

Integrated source code highlighting

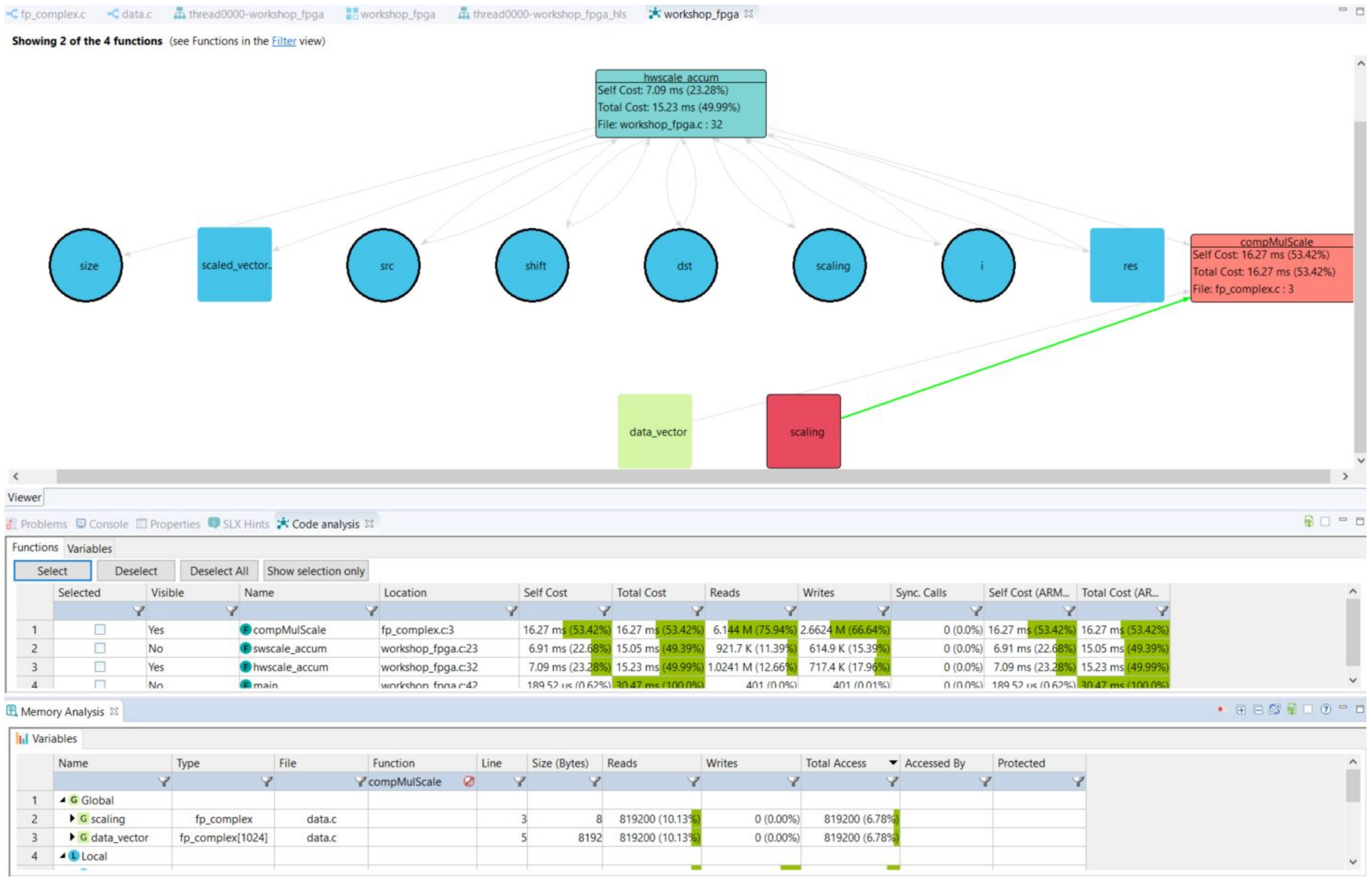
Access statistics

Location

Size

Code Analysis Graph

- Visual representation of relationship between variables and the functions that access them
- Filters allow control over the details in the graph
- Simultaneous variable and function access information



Showing 2 of the 4 functions (see Functions in the Filter view)

hwscale_accum
Self Cost: 7.09 ms (23.28%)
Total Cost: 15.23 ms (49.99%)
File: workshop_fpga.c : 32

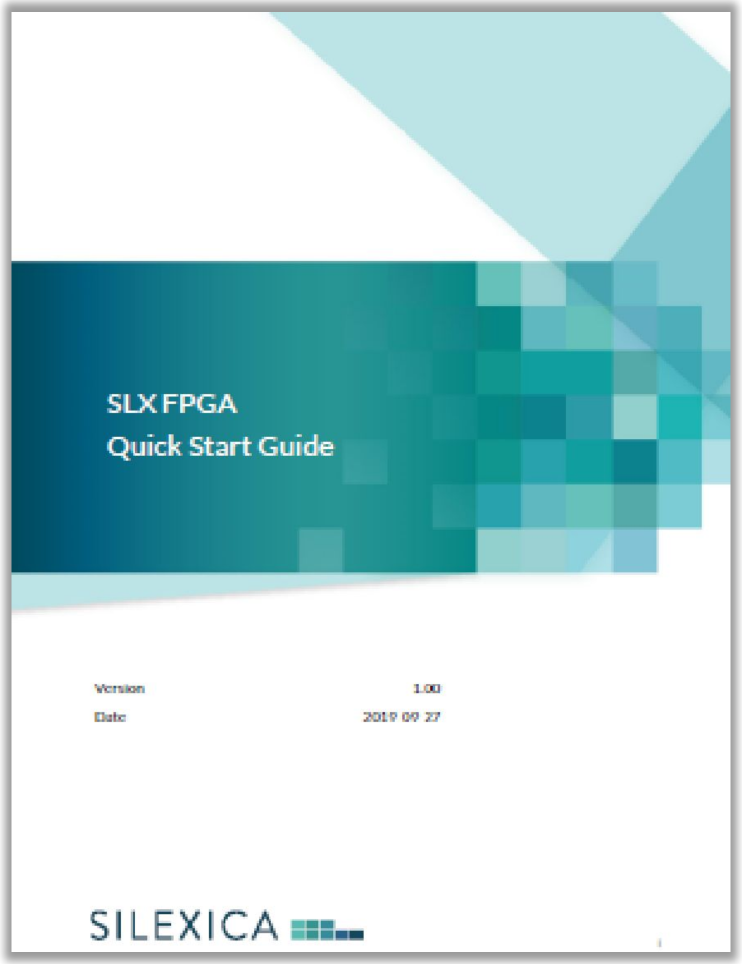
compMulScale
Self Cost: 16.27 ms (53.42%)
Total Cost: 16.27 ms (53.42%)
File: fp_complex.c : 3

	Selected	Visible	Name	Location	Self Cost	Total Cost	Reads	Writes	Sync. Calls	Self Cost (ARM...	Total Cost (AR...
1	☐	Yes	compMulScale	fp_complex.c:3	16.27 ms (53.42%)	16.27 ms (53.42%)	6.144 M (75.94%)	2.6624 M (66.64%)	0 (0.0%)	16.27 ms (53.42%)	16.27 ms (53.42%)
2	☐	No	swscale_accum	workshop_fpga.c:23	6.91 ms (22.68%)	15.05 ms (49.39%)	921.7 K (11.39%)	614.9 K (15.39%)	0 (0.0%)	6.91 ms (22.68%)	15.05 ms (49.39%)
3	☐	Yes	hwscale_accum	workshop_fpga.c:32	7.09 ms (23.28%)	15.23 ms (49.99%)	1.0241 M (12.66%)	717.4 K (17.96%)	0 (0.0%)	7.09 ms (23.28%)	15.23 ms (49.99%)
4	☐	No	main	workshop_fpga.c:42	189.52 us (0.62%)	30.47 ms (100.0%)	401 (0.0%)	401 (0.01%)	0 (0.0%)	189.52 us (0.62%)	30.47 ms (100.0%)

	Name	Type	File	Function	Line	Size (Bytes)	Reads	Writes	Total Access	Accessed By	Protected
				compMulScale							
1	Global										
2	scaling	fp_complex	data.c		3	8	819200 (10.13%)	0 (0.00%)	819200 (6.78%)		
3	data_vector	fp_complex[1024]	data.c		5	8192	819200 (10.13%)	0 (0.00%)	819200 (6.78%)		
4	Local										

Collateral & Real World Results

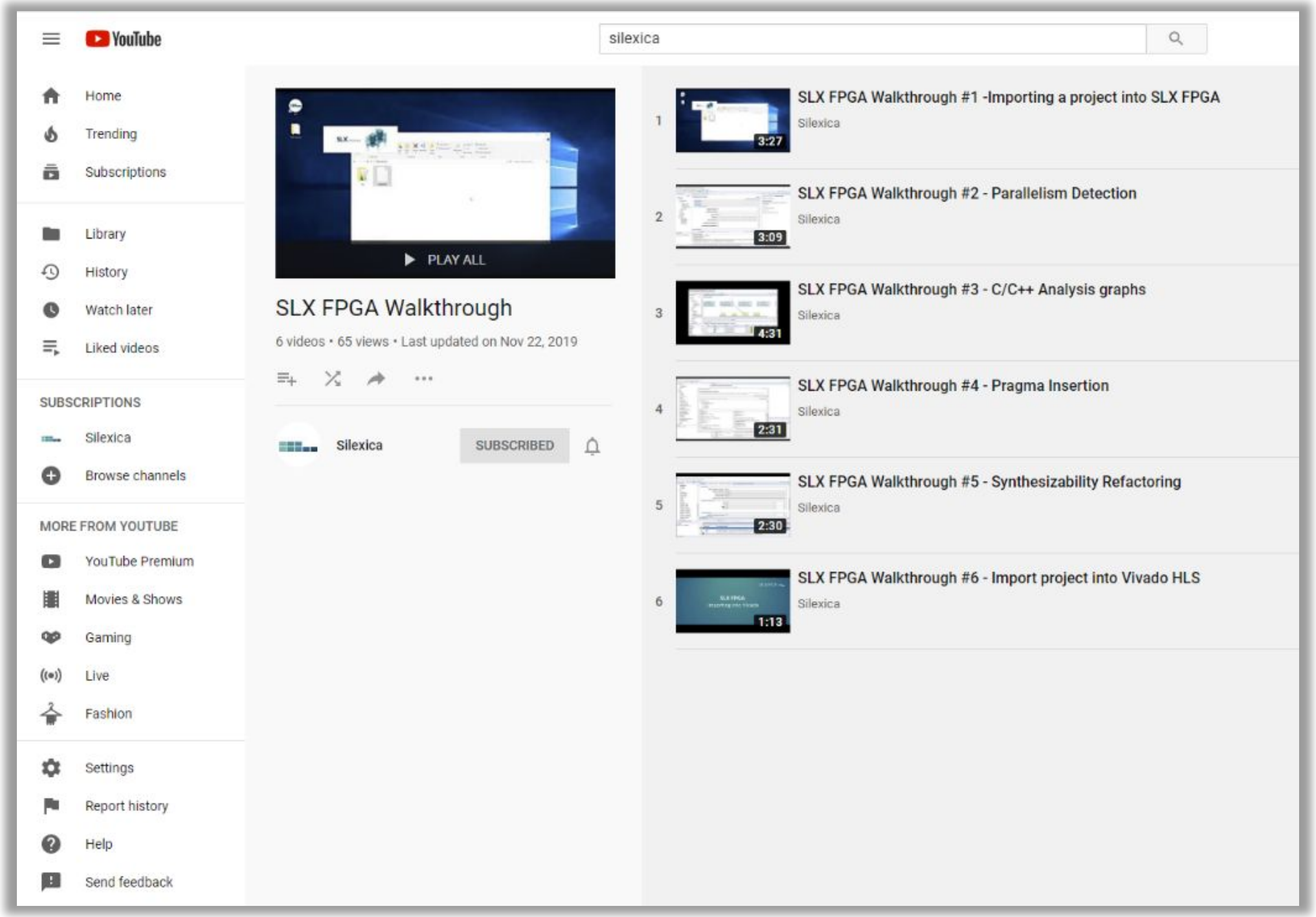
SLX FPGA Getting Started


SLX FPGA
Quick Start Guide

Version 1.00
Date 2019 09 27

SILEXICA

Quick Start Guide



SLX FPGA Walkthrough

6 videos • 65 views • Last updated on Nov 22, 2019

Silexica

1 SLX FPGA Walkthrough #1 -Importing a project into SLX FPGA — Silexica — 3:27

2 SLX FPGA Walkthrough #2 - Parallelism Detection — Silexica — 3:09

3 SLX FPGA Walkthrough #3 - C/C++ Analysis graphs — Silexica — 4:31

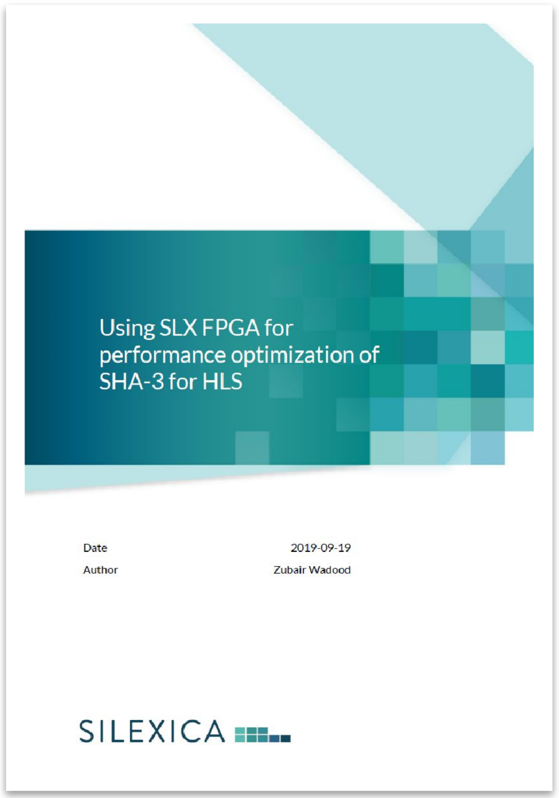
4 SLX FPGA Walkthrough #4 - Pragma Insertion — Silexica — 2:31

5 SLX FPGA Walkthrough #5 - Synthesizability Refactoring — Silexica — 2:30

6 SLX FPGA Walkthrough #6 - Import project into Vivado HLS — Silexica — 1:13

SLX FPGA Walkthrough Videos

SLX FPGA – Application White Papers

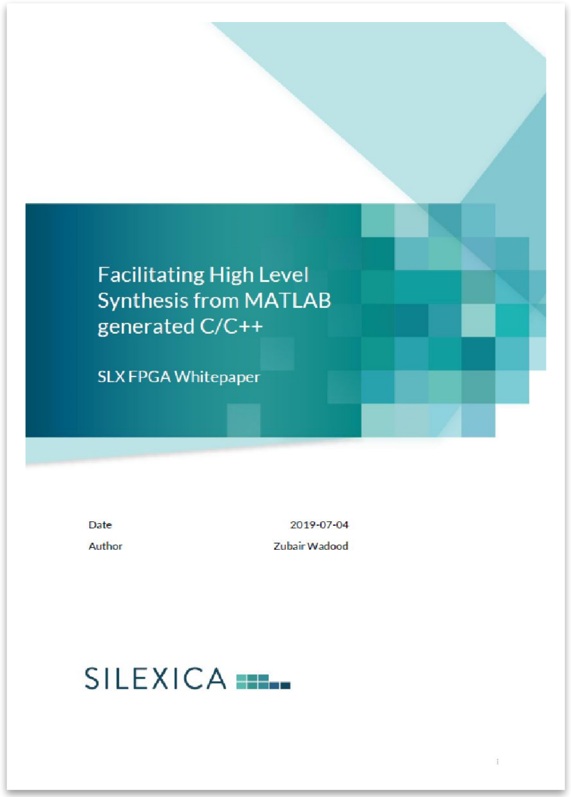

Using SLX FPGA for performance optimization of SHA-3 for HLS

Date 2019-09-19
Author Zubair Wadood

SILEXICA

SHA-3 Algorithm

600x Speed-up with SLX FPGA


Facilitating High Level Synthesis from MATLAB generated C/C++

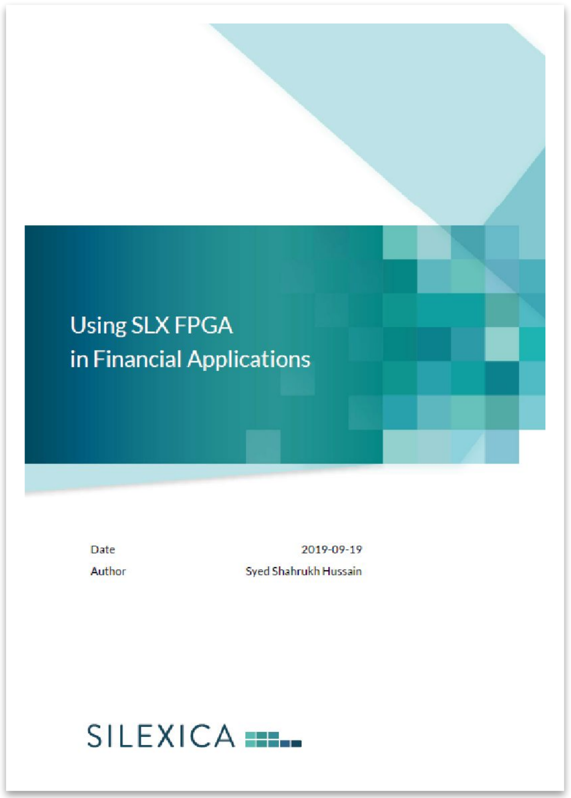
SLX FPGA Whitepaper

Date 2019-07-04
Author Zubair Wadood

SILEXICA

Kalman Filter

62x Speed-up with SLX FPGA


Using SLX FPGA in Financial Applications

Date 2019-09-19
Author Syed Shahrukh Hussain

SILEXICA

Black Sholes & Heston

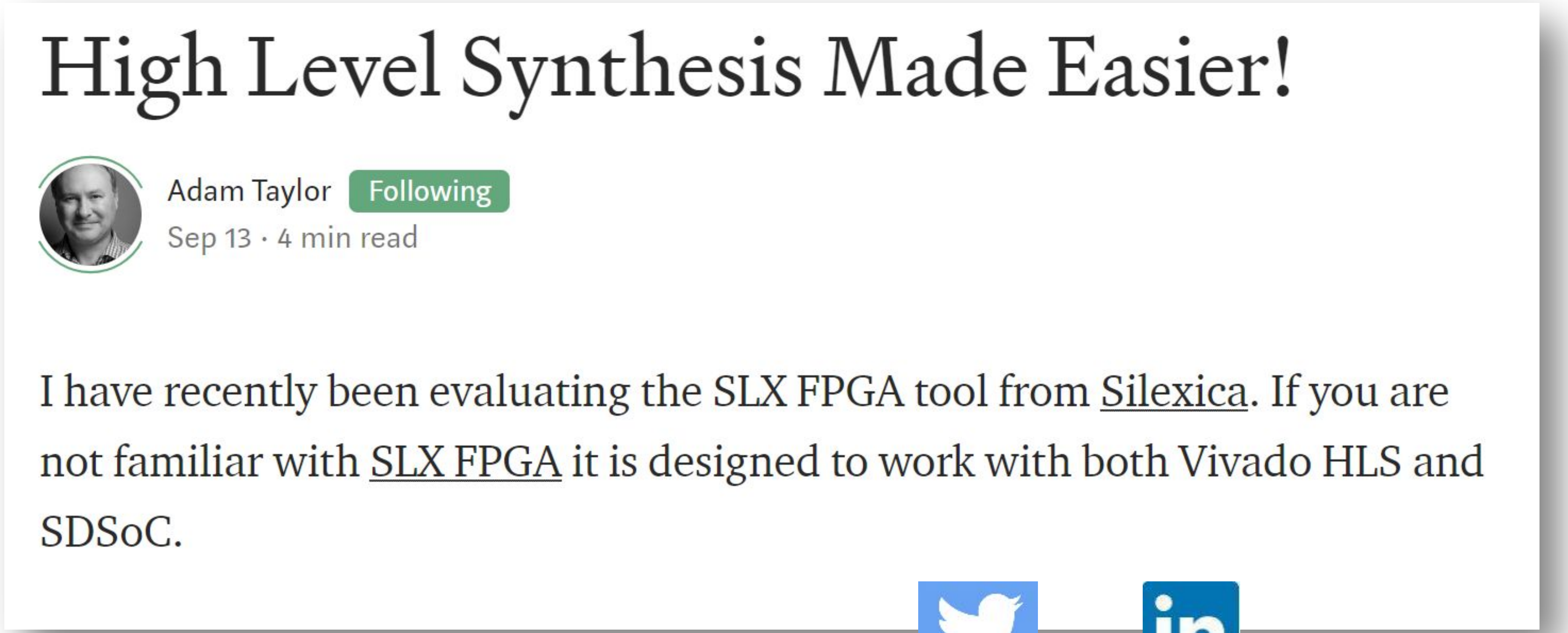
29x Speed-up with SLX FPGA

Reduce Development Time with SLX FPGA

Design Phase	Done by hand (Days)	SLX Optimized (Days)
Clean up code to be synthesizable	1	0.5
Synthesize first HW	1	1
Refine Synthesis by inserting pragmas	10 days	1
Repeat last step until satisfied with results		2
Validate using C/C++ Testbench	0.5	0.5
Create IP	0.5	0.5
Total time	13*	6

- Example from expert HLS user
- Design was relatively small vision processing algorithm (~4KLUTs)
- Final performance achieved with HLS+SLX was better than RTL.

Adam Taylor Blog – Influential Xilinx Blogger



High Level Synthesis Made Easier!

Adam Taylor Following

Sep 13 · 4 min read

I have recently been evaluating the SLX FPGA tool from Silexica. If you are not familiar with SLX FPGA it is designed to work with both Vivado HLS and SDSoC.

“What is interesting to me having worked considerably with HLS over the years is how easy the insertion of pragmas was with SLX FPGA. HLS optimization can be a challenging, iterative and time-consuming process, ***SLX FPGA made this much simpler.****”*

SLX Release Schedule

Release	Quarter	Date
SLX v20.1	Q1	April 6, 2020
SLX v20.2	Q2	June 29, 2020
SLX v20.3	Q3	September 21, 2020
SLX v20.4	Q4	December 14, 2020

Summary

- SLX FPGA accelerates the journey from C/C++ to Hardware by removing many of the roadblocks of using HLS

- SLX FPGA takes the guesswork out of using HLS

- SLX FPGA can help you maximize the performance of your designs in a fraction of the time

SLX FPGA

Silexica is ready to help you get started with SLX FPGA today!