Cologne, Feb 24, 2021

SLX Plugin User Guide
Release 0.9

SILEXICA

Contents

Chapter 1

Introduction

The SLX Plugin is a High-level Synthesis (HLS) compiler add-on for Vitis 2020.2 that helps improve HLS latency and throughput results. Software developers and hardware designers can achieve better results for a set of common design styles and application types with the plugin. When enabled, the plugin augments Vitis HLS' code transformations and adds an additional synthesis directive that helps to squeeze out performance.

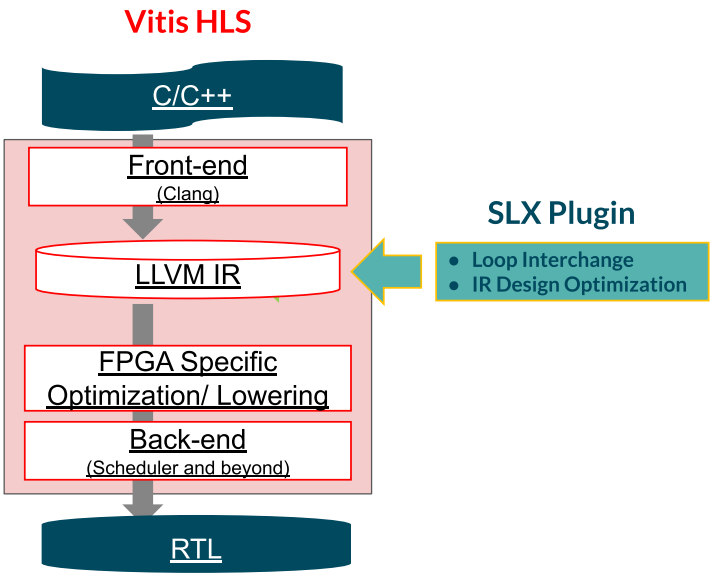

Fig. 1: SLX Plugin for Vitis HLS - Overview

The figure shows how the SLX Plugin directly extends the HLS process for Vitis HLS in two ways. The new synthesis directive, namely `_SLXLoopInterchange`, can be applied to a system of nested loops for performing a loop interchange. Loop interchange exchanges the loop nest iteration variables, effectively interchanging the iteration order in the loop nest. `_SLXLoopInterchange` can be used alone or in combination with other Vitis HLS synthesis directives and constraints to achieve reduced latency and increased throughput. Interchanging iteration orders helps with reordering dependencies that block parallelism, pipelining and memory access patterns. `_SLXLoopInterchange` is described in the *Loop Interchange (_SLXLoopInterchange)* chapter.

The SLX Plugin can be used as a pure standalone addition to Vitis HLS. This way, `_SLXLoopInterchange` directives can be manually inserted into the source code like any other Vitis HLS directive. In the *Quickstart* chapter, we will walk you through the steps to use the SLX Plugin.

As with any other HLS directives and optimization settings, careful analysis is required to determine optimal use of `_SLXLoopInterchange`. To this end, Silexica's standalone SLX FPGA tool can provide tremendous assistance when it comes to optimizing directive use for HLS designs. SLX FPGA makes use of dynamic analysis data and fast performance models to perform automated design space exploration. It enables designers to identify the optimal directives settings for any given set of resource constraints. In addition, its analysis capabilities provide deep insights into the code and data movement to help guide designers whenever code refactoring is necessary. The upcoming version of SLX FPGA, SLX FPGA 2020.4, has been extended to also analyze designs for opportunities to apply the `_SLXLoopInterchange` directive whenever it deems that performance will improve as a result. This allows using SLX FPGA, the SLX Plugin and Vitis HLS together for attaining an improved HLS development methodology. See the how to *Use with SLX FPGA* chapter for more information.

Finally, the SLX plugin also provides a few automatic compiler transformations that complement transformations performed by Vitis HLS. In certain key scenarios, an HLS design's internal program representation can be improved, leading to improved HLS design performance. These transformations are collectively called IR design optimizations. They are described in the *IR design optimizations* chapter of this user guide.

Chapter 2

Quickstart

Follow these instructions to quickly start using the SLX Plugin. The instructions explain how to execute the provided examples and interpret the results.

2.1 Setup

The SLX Plugin is currently supported for Ubuntu 18.04. Make sure Vitis HLS 2020.2 is installed in your machine. After untarring the plugin, execute the following command from within the plugin's root directory:

```
source exports
```

Run the command in the same console where you will later run Vitis HLS. The shell needs to be configured to run the Vitis tools.

2.2 Exploring the examples

The SLX Plugin is shipped with several examples. Examples include C/C++ source code, scripts and reference results files. For a quick start, go to an example folder and execute the provided `check.sh` script. For example, perform the following to start with the `dimension_reduction` example:

```
cd examples/dimension_reduction/reduce_2d_to_1d
./check.sh
```

The script will call Vitis HLS and synthesize the design. The script will generate different solutions using different features of the SLX Plugin. For example, there will be solutions with and without loop interchange. Co-simulation is not run by default, but can be enabled by editing `x_hls.tcl` and setting `hls_exec` to 2.

Key values are extracted from Vitis HLS solution reports and captured in a newly created `latency.ref` file. If the results differ from the ones saved in `latency.ref` you'll get a message. A different file called `latency.new` will be created. This could happen if you use a different Vitis HLS version from the one used when `latency.ref` was created.

2.3 Investigating the results

Almost all examples generate the following three solutions:

- `vanilla`: uses Vitis HLS off-the-shelf without any SLX Plugin additions.
- `nointerchange`: applies *IR design optimizations*. This can already change results for some examples, but it doesn't interchange the loops.
- `interchange`: applies `_SLXLoopInterchange` directives inserted in the example source files by interchanging the specified loop nests.

Some examples generate a few more results because, for example, they generate additional solutions to illustrate additional effects on the design, such as the effect of different target clock constraints. This is the case for `dimension_reduction`.

Looking at the tables in `latency.ref` file helps in comparing some key results. The Vitis HLS project that is also produced can be opened in the Vitis HLS GUI for more detailed inspection. The following is an excerpt of `latency.ref` for the `dimension_reduction` example. The table shows changes in the hotspot loop's latency and initiation interval. When comparing `vanilla` to `interchange`, iteration latency goes from 1800 to 269 when measured at the outer loop and initiation interval goes from 7 to 1 for the inner loop. This means that `interchange` achieves an improvement of ~7x.

```
## 0.vanilla: no LLVM_CUSTOM_CMD, no directives (clock: 333MHz)
|               |  Latency (cycles)  | Iteration |  Initiation Interval  |
|   Loop Name   |    min   |   max   |  Latency  | achieved |   target   |
|- loop_out     |   460800 |  460800 |      1800 |        - |          - |
|  `- loop_in   |     1796 |    1796 |        12 |        7 |          1 |

## 2.interchange: LLVM_CUSTOM_CMD -slx-loop-interchange, no other  directives (clock:
↪333MHz)
|               |  Latency (cycles)  | Iteration |  Initiation Interval  |
|   Loop Name   |    min   |   max   |  Latency  | achieved |   target   |
|- loop_in      |    68863 |   68863 |       269 |        - |          - |
|  `- loop_out  |      266 |     266 |        12 |        1 |          1 |
```

Interchanging moves a key loop carried dependency from the inner to the outer loop. With that done, Vitis HLS is free to pipeline the inner loop with a far better initiation interval (II).

Whether a loop carried dependency plays a key role depends on the latency of the operations needed to compute a new value from the old value. The `dimension_reduction` example explores this by comparing the design at 333MHz and 85MHz. At the lower frequency, the operation completes within a cycle and the dependency no longer plays a key role–this is visible in `latency.ref` as interchanging does not help this design at 85MHz.

2.4 Using the loop interchange directive

This section explains how to enable the SLX Plugin and apply `_SLXLoopInterchange` quickly to start with your own design. For using the SLX Plugin in an existing project, a couple of additional Tcl commands should be added to your design's Tcl files to steer Vitis HLS - see *Existing Vitis HLS Command Line Project (Tcl)* or *Existing Vitis HLS GUI Project* for details. In `dimension_reduction` `run_hls.tcl` the additions are:

```
# 1. Variable pointing at the plugin location
set slxplugin [file normalize $::env(SLX_VITIS_PLUGIN_HOME)]
```

(continues on next page)

(continued from previous page)

```
# 2. Include slxplugin.h when compiling the design file
add_files code.cpp -cflags "-include $slxplugin/include/slxplugin.h"

# 3. Add slxplugin.so using Vitis HLS injection mechanism
#    - $slx_options are solution-specific options, and -slx-remove-directives
#      is to remove the directive if it is not being used for a solution
set ::LLVM_CUSTOM_CMD [concat \
    {$LLVM_CUSTOM_OPT -load} $slxplugin/lib/slxplugin.so \
    $slx_options -slx-remove-directives \
    {$LLVM_CUSTOM_INPUT -o $LLVM_CUSTOM_OUTPUT}]
```

Afterwards, using the loop interchange directive in the source code is very simple. Just add `_SLXLoopInterchange();` (note the C function call style syntax[1]) to a loop nest, under the loop level that you want to interchange with its immediate parent loop. The following code shows how this is done for a loop nest composed of the outer `loop_out` and the inner `loop_in` loops. Since the directive is added under the latter, the transformation will make `loop_in` the outer loop and `loop_out` the inner loop. This will effectively transform the `i then j` iteration order into `j then i`.

```
loop_out:for (int i = 0; i < N; i += 1) {
  loop_in:for (int j = 0; j < N; j += 1) {
    _SLXLoopInterchange();
    acc[i] = acc[i] + data[i][j];
  }}
```

[1] The C function call style syntax will be modified in a future release to be syntactically closer to Xilinx HLS pragmas.

Chapter 3

Instructions

This section provides installation steps and instructions to use the SLX Plugin in stand-alone mode with Vitis 2020.2. For instructions to use the plugin with SLX FPGA, refer to the SLX FPGA User Guide after following the steps in the *Installation* section.

3.1 Supported machines and requirements

The SLX Plugin currently supports[1] Ubuntu 18.04 on x86_64 architectures. The next steps assume that Ubuntu 18.04 has been installed on your machine. Since the SLX Plugin is an add-on for Vitis 2020.2, these steps also assume that Vitis is properly installed on your machine.

3.2 Package contents

Untar the provided SLX Plugin package. This should result in the following folder structure:

```
.
└── slxplugin-x.y.z
    ├── doc
    ├── examples
    ├── exports
    ├── include
    │   └── slxplugin.h
    ├── lib
    │   └── slxplugin.so
    └── README
```

The contents are as follows:

- *doc*: the SLX Plugin documentation in html and pdf format.
- *examples*: example designs that benefit from the SLX Plugin transformations and show the plugin in action.
- *exports*: setup script. Source this script every time you want to use the plugin.
- *include*: the SLX Plugin header file to be added via a Vitis HLS Tcl script to activate the plugin.

[1] The SLX Plugin is only officially supported on Ubuntu 18.04 on x86_64 architectures. However, it is also known to work with Ubuntu 16.04. Try Ubuntu 16.04 at your own risk.

- *lib*: the SLX Plugin library file to be added via a Vitis HLS Tcl script to activate the plugin.
- *README*: file with quick pointers and guidelines.

3.3 Installation

Once the SLX Plugin is unpacked, no further installation is required. Just execute the following command in your Linux console. This allows our example scripts and SLX FPGA to locate the plugin files in your machine.

```
source exports
```

This will set the environment variable `SLX_VITIS_PLUGIN_HOME` to the location of your SLX Plugin package.

Attention: `SLX_VITIS_PLUGIN_HOME` is only set in the console session where the command is executed. Execute it in the console where you will execute Vitis HLS. If you open a new console, remember to execute the command again.

Tip: Add the command to your `.bashrc` if you want to automatically set `SLX_VITIS_PLUGIN_HOME` every time a new console is opened.

Afterwards, the SLX Plugin can be injected into the Vitis HLS compilation flow by adding just a few Tcl commands to an *Existing Vitis HLS Command Line Project (Tcl)* or an *Existing Vitis HLS GUI Project*. If you use the SLX Plugin with SLX FPGA, read how to use the plugin with an *SLX FPGA Project*.

3.4 Existing Vitis HLS Command Line Project (Tcl)

If your Vitis HLS project is a command line project based on Tcl, you can enable the plugin by adding the following commands to your script:

```
# 1. Variable pointing at the plugin location
set slxplugin [file normalize $::env(SLX_VITIS_PLUGIN_HOME)]

# 2. Add slxplugin.h
#   - <your-design-files> refers to your C/C++ files
add_files <your-design-files> -cflags "-include $slxplugin/include/slxplugin.h"

# 3. Add slxplugin.so using Vitis HLS injection mechanism
#   - <slx-plugin-options> refers to options provided by SLX Plugin
set ::LLVM_CUSTOM_CMD [concat \
    {$LLVM_CUSTOM_OPT -load} $slxplugin/lib/slxplugin.so \
    <slx-plugin-options> \
    {$LLVM_CUSTOM_INPUT -o $LLVM_CUSTOM_OUTPUT}]
```

These three commands will activate the SLX Plugin for your design. They perform the following actions:

1. The first command sets a variable that points to the plugin installation directory.

2. The second command adds the plugin header file that enables Vitis HLS compilation with the SLX Plugin directives. Replace `<your-design-files>` with your actual C/C++ files that contain additional SLX Plugin directives.
3. The third command injects the plugin library file into the Vitis HLS synthesis process. This allows performing the transformations provided by the SLX Plugin. Replace `<slx-plugin-options>` with the SLX Plugin options for the current solution. For example, replace it with `-slx-prepare-for-interchange -slx-loop-interchange` when using the plugin with the `_SLXLoopInterchange` directive. `csynth_design` applies these options during synthesis. Options currently supported by the SLX Plugin are described in the *Options reference guide* section.

3.5 Existing Vitis HLS GUI Project

If you have an existing Vitis HLS GUI project, you need to perform two steps:

- in the GUI, update the `CFLAGS` in `Project` -> `Project Settings` dialog, under the Synthesis tab. Add `-include $::env(SLX_VITIS_PLUGIN_HOME)/include/slxplugin.h` as shown in the picture:

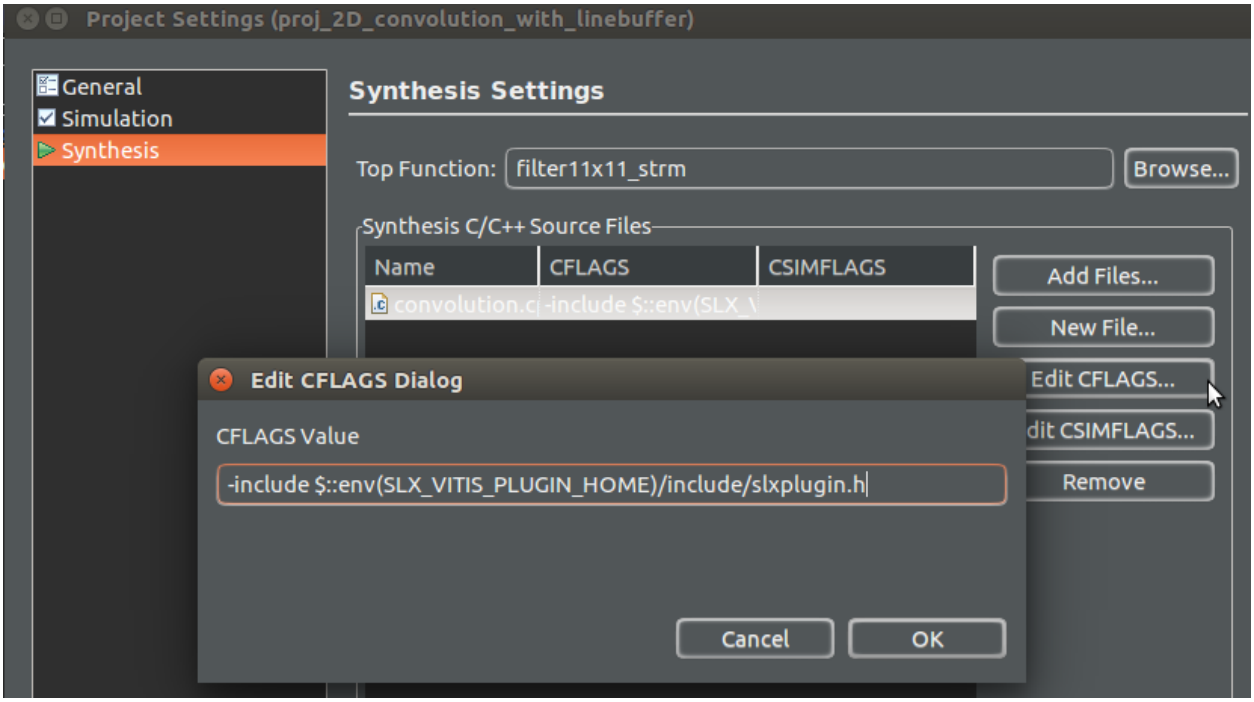

Fig. 1: Include the SLX Plugin header in CFLAGS.

- Add the code below to `scripts/vitis_hls_init.tcl`. You can create the file in the GUI using `File` -> `New File`. If the `scripts` folder is not yet present, click the `Create Folder` icon to add it. Create the `vitis_hls_init.tcl` in that folder. The file opens in the GUI but is not editable there. Use `File` -> `New File` again to open the dialog and go to the `scripts` folder. Right-click on `vitis_hls_init.tcl` and `Copy location` to get the path. Please then open the file with your favorite text editor to add the code:

```
# add slxplugin.so using Vitis HLS injection mechanism
set ::LLVM_CUSTOM_CMD [concat \
```

(continues on next page)

(continued from previous page)

```
    {$LLVM_CUSTOM_OPT -load} $::env(SLX_VITIS_PLUGIN_HOME)/lib/slxplugin.so \
    <slx-plugin-options> \
    {$LLVM_CUSTOM_INPUT -o $LLVM_CUSTOM_OUTPUT}]
```

These are adaptations of the commands already explained in the previous section for using the SLX Plugin with an *Existing Vitis HLS Command Line Project (Tcl)*. Make sure that you are modifying the file that corresponds to your active solution, and remember to adapt `<slx-plugin-options>` as explained before.

3.6 Main option combinations

The most useful sets of SLX Plugin options to use as `<slx-plugin-options>` are:

- To use *Loop Interchange (_SLXLoopInterchange)*, enable with:

```
-slx-prepare-for-interchange -slx-loop-interchange
```

  - To have the `_SLXLoopInterchange()` directive in the source code but skip its application or ignore a failure to apply it: add `-slx-remove-directives` to the previous set.

- To try out *IR design optimizations*, enable with:

```
-slx-prepare-for-interchange -slx-prepare-all
```

The options are explained at the end of this chapter in the *Options reference guide* section.

3.7 Loop Interchange (`_SLXLoopInterchange`)

The SLX Plugin augments Vitis HLS with an additional optimization directive for achieving higher performance. The new synthesis directive is called `_SLXLoopInterchange` and can be applied to a system of nested loops for performing a loop interchange. Loop interchange is a transformation that operates by exchanging the order of two iteration variables in the loop nest, effectively interchanging the iteration order in the loop system.

`_SLXLoopInterchange` is used in the source code like any other Xilinx HLS pragma. When added to a loop that is part of a loop nest, the SLX Plugin will interchange the target loop and its immediate parent. The target loop is defined as the loop that contains `_SLXLoopInterchange` within its direct body's scope. The directive is using the C function call style syntax[2] as shown in the next few examples. The `<slx-plugin-options>` to apply this directive are `-slx-prepare-for-interchange -slx-loop-interchange`.

First, consider the code excerpt shown below:

```
LoopN: for ( int i = 0; i < n; i++ ) {
LoopM:   for ( int j = 0; j < m; j++ ) {
           _SLXLoopInterchange();
           out[i][j] = compute( in[i][j] );
       }}
```

It has a nested loop composed of `LoopN` (the inner loop) and `LoopM` (the outer loop). The `_SLXLoopInterchange` directive is added under `LoopM`, which turns into the target to be interchanged with its immediate parent. Since `LoopM` has only one parent, the action leads to interchange `LoopM` with

[2] The C function call style syntax will be modified in a future release to be syntactically closer to Xilinx HLS pragmas.

LoopN. LoopM will be transformed into the outer loop and LoopN into the inner one. This will effectively transform the i then j iteration order into j then i.

For the following code example, things work as follows:

```
LoopO: for ( int i = 0; i < o; i++ ) {
LoopN:   for ( int j = 0; j < n; j++ ) {
           _SLXLoopInterchange();
LoopM:     for ( int k = 0; k < m; k++ ) {
             out[i][j][k] = compute( in[i][j][k] );
       }}}
```

_SLXLoopInterchange is added under LoopN, which turns into the target to be interchanged with LoopO. This will effectively transform the i then j then k iteration order into j then i then k. Similarly, _SLXLoopInterchange could be applied to LoopM if the directive is moved two lines down, which would in turn switch the iteration order to i then k then j.

The loop interchange transformation has a good impact on a design's performance when by interchanging iteration orders one relaxes conditions that limit parallelism or pipelining. If this is possible for a hotspot loop, then one can achieve a good improvement in initiation intervals and latency. The following code excerpt shows a situation that benefits from _SLXLoopInterchange:

```
LoopN:  for(int i = 0; i < n; i += 1) {
LoopM:    for(int j = 0; j < m; j += 1) {
            _SLXLoopInterchange();
            acc[i] = acc[i] + data[i][j];
        }}
```

In the code there is a loop carried dependency on variable acc in the inner LoopM that blocks potential pipelining with II=1 under certain circumstances. Interchanging the loop moves the dependency from the inner to the outer loop LoopN, effectively relaxing the constraint that blocks achieving pipeline with II=1. For more information about this example and other cases that benefit from _SLXLoopInterchange, refer to the *Examples*

_SLXLoopInterchange can be used alone or in combination with other Vitis HLS synthesis directives and constraints to achieve reduced latency and increased throughput. When this happens, other Vitis HLS loop directives in the same loop of a _SLXLoopInterchange migrate to the new loop level after the interchange is performed. This means that the following code:

```
LoopN: for ( int i = 0; i < n; i++ ) {
LoopM:   for ( int j = 0; j < m; j++ ) {
           _SLXLoopInterchange();
           #pragma HLS pipeline
           out[i][j] = compute( in[i][j] );
       }}
```

will be transformed to:

```
LoopM: for ( int j = 0; j < m; j++ ) {
         #pragma HLS pipeline
LoopN:   for ( int i = 0; i < n; i++ ) {
           out[i][j] = compute( in[i][j] );
       }}
```

Attention: If the option -slx-loop-interchange or -slx-remove-directives is not used, Vitis HLS will fail during synthesis with an error: WARNING: [HLS 200-412]

`1_VALIDATOR__call_node_lacks_subCdfg` and `Stack dump: Running pass 'CDFG Construction Pass' on module 'a.o.2.bc'.`. Please check that the `LLVM_CUSTOM_CMD` is being set properly - this should appear in the Vitis HLS console output.

3.7.1 Transformation safety and legality

Performing a loop interchange is not always safe or legal. Exchanging the iteration order of a loop nest could sometimes lead to an invalid computation. The SLX Plugin incorporates a layer to detect transformation safety and legality and rejects it when it is certain the interchange can't be applied. The following rules apply:

- Loop interchange is limited to perfect loop nests. This limitation will be relaxed in a future release.
- Dependency inversion issue: the transformation is rejected if by interchanging you will compute a value either earlier or later compared to its original use point. For example, you are reading from a row that is only updated in the future to compute a value that is needed in the present. `SLXLoopInterchange` checks for legality in this case and rejects the transformation.
- Floating point associativity issue: floating point computation is not necessarily associative. That is, `(a + b) + c` is not necessarily equal to `a + (b + c)`. In many cases, this only leads to minor precision differences. `SLXLoopInterchange` won't try to reason about the user's intentions from this perspective. The suggested method is to address this via testbenches. If your testbench fails, disable `SLXLoopInterchange`.
- Loop flatten conflict: when applied, `SLXLoopInterchange` disables the Vitis HLS loop flatten transformation. By default, loop flatten is automatically applied by Vitis HLS to perfect or semi-perfect loop nests. Additionally, it can be controlled with `set_directive_loop_flatten` in a Tcl directives script or with `#pragma HLS loop_flatten` in the C/C++ source code. If a `_SLXLoopInterchange` is applied to the same loop then a warning is reported:

```
``WARNING: [SLX] HLS flatten for '<loop>'(<location>) is disabled as the loop is␣
↪transformed into a do-while loop``
```

- Loops composed of statically unknown bounds are not supported. If the safety check is not able to tell for certain that an access will be properly bounded when accessing the array, the loop gets rejected. In the following example, it is not possible to tell that `m<=10`. Because `a[0][10]` addresses the same element as `a[1][0]` in the C memory model, there is a possibility that changing the iteration order will cause some data to be re-written by out-of-bound accesses. For this reason, it is forbidden.

```
int a[10][10];
void test(int n, int m) {
 for (int i = 0; i < n; ++i)
   for (int j = 0; j < m; ++j)
     a[i][j] = j; // This access is causing an issue
}
```

3.8 IR design optimizations

The SLX Plugin contains a few automatic compiler transformations that help improve an HLS design's internal program representation before the RTL code is generated. The most interesting scenario is when it helps to booster Vitis HLS' capabilities for automatic relocation of loop invariant memory accesses. While this doesn't apply to all HLS designs, some designs benefit from improved results when, for example, expensive memory accesses are manipulated in a way that they end up outside of a loop nest. In particular, if the loop nest

happens to be a computation hotspot, this can be of tremendous help to improve latency and throughput without manual C/C++ source code changes.

This feature doesn't require adding directives to source code. It is enabled by using the options `-slx-prepare-for-interchange` and `-slx-prepare-all` in the Tcl command that injects the SLX Plugin to the Vitis compilation process. If both options are specified, then IR design optimizations are applied to all functions in the design. If `-slx-prepare-all` is not specified, then IR design optimizations will only apply to functions that contain an `_SLXLoopInterchange` directive in one of their loops. For more information about the Tcl command, see *Existing Vitis HLS Command Line Project (Tcl)*. For more information about the options, see the *Options reference guide*.

An example that shows and explains an IR design optimization is provided in *Examples*. Refer to the `tiny-sha3` folder.

3.9 Options reference guide

The SLX Plugin currently supports the following options that control its behavior:

- `-slx-prepare-for-interchange`: perform IR design optimization passes and enablers for loop interchange. By default, this only applies for functions calling `_SLXLoopInterchange()`. To change the default behavior, see `-slx-prepare-all`. It reports in the console if it knows it will result in a Xilinx HLS pragma not applying:

```
``WARNING: [SLX] HLS flatten for '<loop>'(<location>) is disabled as the loop is␣
↪transformed into a do-while loop``
```

- `-slx-prepare-all`: only valid in combination with `-slx-prepare-for-interchange`. When given, IR design optimization passes are applied to all functions and not only those containing `_SLXLoopInterchange()`.
- `-slx-loop-interchange`: perform loop interchange if possible. By default, only do so for functions calling `_SLXLoopInterchange()`. It reports in the console if the interchange was successful:

```
``SLX-INFO: _SLXLoopInterchange applied successfully``
``SLX-ERROR: _SLXLoopInterchange was requested but was not successfully applied``
```

- `-slx-fatal-unapplied-interchange`: only valid in combination with `-slx-loop-interchange`. The default is to issue an `ERROR` if the `_SLXLoopInterchange()` is not applied, but when given with the `=FALSE` suffix a non-fatal `WARNING` is produced instead.
- `-slx-remove-directives`: delete `_SLXLoopInterchange()`. This is useful if the current solution should not include loop-interchange, or if loop-interchange was not possible and you wish to ignore this.

For example, if all options are applied, the last command in the setup Tcl script will be:

```
# add slxplugin.so using Vitis HLS injection mechanism
set ::LLVM_CUSTOM_CMD [concat \
    {$LLVM_CUSTOM_OPT -load} $slxplugin/lib/slxplugin.so \
    -slx-prepare-for-interchange \
    -slx-prepare-all \
    -slx-loop-interchange \
    -slx-remove-directives \
    {$LLVM_CUSTOM_INPUT -o $LLVM_CUSTOM_OUTPUT}]
```

Chapter 4

Examples

This chapter describes the code shipped in the `examples` folder. The examples illustrate how loop interchanging can improve the performance of a design, and how the SLX Plugin can be used within Vitis HLS to perform the interchange.

4.1 Dimension reduction (`dimension_reduction`)

Dimension reduction combines information from multiple dimensions to a smaller number, typically as a lossy operation resulting in denser information. Dimensionality reduction is common in fields that deal with large numbers of observations and/or large numbers of variables, such as signal processing, speech recognition, neuroinformatics, and bioinformatics

In `code.cpp` we see the standard way to write a 2D to 1D reduction in software.

```
loop_out:for (int i = 0; i < N; i += 1) {
    loop_in:for (int j = 0; j < N; j += 1) {
        acc[i] = acc[i] + data[i][j];
}}
```

In the inner loop, the `acc[i] = acc[i] + ...` means that the `acc[i]` computed in the previous iteration has to be available, and the `+` operation has to produce a new `acc[i]` value in the current iteration that will be stored in memory. As the operation has to complete before the operation in the next cycle can begin, its latency always contributes to the II of the loop. If it is the slowest operation contributing, then it is the operation that extends the II.

This can be seen when different clock frequencies are used. For example, when the clock is set to 85MHz an `II=1` is achieved because the `+` then completes within a cycle, alongside the other contributors to the II. As the clock frequency goes up, the operation becomes multicycle. At 333MHz this increases up to `II=7`. This can be see by comparing result tables `0.vanilla` (333MHz) and `3.vanilla` (85MHz) in `latency.ref`.

This is where `_SLXLoopInterchange` helps. If the inner and outer loop iteration is swapped, the same result is produced but the operation schedule and corresponding timing is very different. By doing this, the `acc[i]` value being read is that produced from the previous time the inner loop in its entirety executed. This means the `+` operation no longer has to complete within the II of the inner loop. At both 85MHz and 333MHz an `II=1` is achieved when the loops are interchanged. This can be see by comparing result tables `2.interchange` (333MHz) and `5.interchange` (85MHz) in `latency.ref`.

In sum, the fact that there is a loop carried dependency in the inner loop, the iteration order of the loops, the latency of involved operations, and a high frequency set conditions for having a bad II. However, by

using `_SLXLoopInterchange` the loop carried dependency is removed from the inner loop and an optimal pipeline `II=1` is achieved.

4.2 DFT and variants (`dft`)

The discrete Fourier transform (DFT) is perhaps the most essential tool in signal processing. It is used to compute a signal's frequency spectrum and a system's frequency domain response. This allows reflecting on and analyzing information encoded in a signal's frequency, phase and amplitude as well as on a system's capabilities to react to such type of signals. There are several ways to implement a DFT, including the versatile and computation efficient fast Fourier transform (FFT) which is a basic block for many applications in wireless telecommunications, audio processing, radars, among many others.

The example in this folder is a non-optimized DFT implementation, written in the style that closely matches the algorithm specification:

```
loop_freq:for (i = 0; i < N; i += 1) {
    temp_real[i] = 0;
    temp_imag[i] = 0;
    // (2 * pi * i)/N
    w = (2.0 * 3.141592653589 / N) * (TEMP_TYPE)i;
    // Calculate the jth frequency sample_sequentially
    loop_calc:for (j = 0; j < N; j += 1) {
        c = cos(j * w);
        s = sin(j * w);
        // Multiply the current phasor with the appropriate input sample
        // and keep a running sum
        temp_real[i] += (sample_real[j] * c - sample_imag[j] * s);
        temp_imag[i] += (sample_real[j] * s + sample_imag[j] * c);
    }
}
```

The inner loop contains two important loop carried dependencies, `temp_real[i] +=` and `temp_imag[i]`, of floating-point values. When pipelined, these addition operations contribute to the II of the inner loop, and if the addition is a multi-cycle operation then the II will also be multicycle - see the related discussion in the *Dimension reduction (dimension_reduction)* example. The application will then benefit from `_SLXLoopInterchange` because the loop carried dependencies will then move to the outer loop and no longer contribute to the II of the inner loop. This is shown in the two `dft` variants provided that are provided in the package.

The effect of relocating the loop carried dependency is better seen in the `perfectized` example. This example is the result of applying a loop perfectization transformation to the code excerpt shown above. This way, `_SLXLoopInterchange` can be applied successfully, as can be seen in `latency.ref` where `loop_calc` is now the reported outer loop. Comparing the non-interchanged `1.nointerchange` inner loop `II=9` to the `2.interchange` inner loop `II=1` shows the benefit of performing the interchange.

The `orig` example is the original software implementation from the code excerpt. This example leads to a couple of interesting effects with the SLX Plugin. On the one hand, the loops are not perfectly nested so applying an interchange is not possible without perfectizing the loop nest. On the other hand, the application of IR design optimization options changes the code structure so that other Vitis HLS transformations kick in. These transformations also help to achieve a good `II=1` in spite of the loop carried dependency. This is visible in the `latency.ref` results where both `1.nointerchange` and 2.interchange achieve `II=1`. Note that `2.interchange` reports the loops in their non-interchanged order, meaning that the interchange transformation was not performed.

4.3 BOTDA: Brillouin optical time domain analyzer (botda)

BOTDA sensors are used for temperature and strain monitoring of oil & gas pipelines, bridges, dams, security fences and power lines. They are excellent for detecting corrosion, buckling, down hole, micro cracks, and leakages in pipelines. It can also be used for fire detection and improve efficiency in oil refineries.

The example contains a linear support vector regression (LSVR) function that is the algorithm's hotspot. SVR and LSVR are based on matrix-vector multiplication, which makes them good candidates for pipelining and parallelization from a hardware perspective.

The code can be found in `botda.cpp`. The hotspot function `linearSVR()` implements the SVR algorithm that aims to apply the regression to the data. Two versions of the algorithm, `orig` and `split` are available. `orig` implements a naive version of the algorithm, `split` implements a version where the main loop nest has been divided in order to reduce the inner loop carried dependencies.

The existence of a third loop `L3` in this second example does not reduce performance, because it allows for the main loop nest to be perfectly nested, thus providing more efficient optimizations. However, even after splitting, the hotspot still has an inner loop carried dependency `partial_sum[i] +=` that contributes to an `II=9`. Loop interchange relocates the dependency to the outer loop, allowing for an improved `II=1` and an improved loop nest latency through more efficient pipelining.

4.4 MRI reconstruction (mri/reconstruction)

MRI reconstruction transforms the raw data acquired by an MRI scanner into images that can be interpreted clinically. There are several methods of scanning and interpreting the results. The algorithm included here performs an anatomically constrained image reconstruction of non-Cartesian (spiral) scan data which has only recently been made possible due to its high computational intensity. The design is based on a conjugate gradient algorithm which determines the output voxels iteratively to improve the value of a cost function. Our example accelerates the first step of the three steps in this algorithm. This step calculates a convolution kernel and is the most computationally intensive part of the algorithm since it traverses the outer loop `8 * N` times. Interchanging the two loops removes the dependency when accumulating the output data in the destination arrays `rQ` and `iQ` from the inner loop. This allows for an `II=1` compared to an `II=5` in the original code.

4.5 SHA-3 algorithm (tiny_sha3)

SHA stands for secure hash algorithm. It used for creating secure hashing of data and certificate files. Every piece of data produces a unique hash that is thoroughly non-duplicable by any other piece of data. `tiny_sha3` is a small implementation of the FIPS 202 and SHA3 hash function.

In `sha3.c` is where the heavy computation happens. The hotspot starts at the following loop header in function `sha3_keccakf`:

```
void sha3_keccakf(uint64_t st[25]) {
    ...
    for (r = 0; r < KECCAKF_ROUNDS; r++) {
        for (i = 0; i < 5; i++)
            bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];
        for (i = 0; i < 5; i++) {
            t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
            for (j = 0; j < 25; j += 5)
                st[j + i] ^= t;
```

(continues on next page)

(continued from previous page)

```
        }
        ...

    }
```

Vitis HLS automatically applies the HLS `pipeline` directive to that loop and so all inner loops are fully unrolled. Other HLS compiler optimizations are performed, resulting in 25 values being loaded from memory for variable `st` once per iteration, near the start of the iteration. Then near the end of the iteration 25 values are stored to `st`. All intermediate stores are replaced by internal local store of data that is implemented using flip-flops (FFs).

When the SLX Plugin's IR design optimizations are applied, Vitis HLS sees that it can further load values from `st` into FFs once before the loop starts, and replace all stores in the loop body by stores that occur after the loop completes. It can do this as it is certain that the loop body will be executed at least once. This means that the loop body does not contain any loads or stores for `st` anymore, and so can be implemented in highly parallelizable logic This results in `II=1` and `latency=24` for the loop compared to `II=26` and `latency=624` when not using the SLX Plugin.

Chapter 5

Use with SLX FPGA

SLX FPGA is a tool designed to accelerate HLS design flows. It tackles the challenges associated with the HLS design flow including non-synthesizable C/C++ code, non-hardware aware C/C++ code, detecting application parallelism, and where and how to insert pragmas. The figure below gives an overview of the SLX FPGA workflow for implementation and optimization of an HLS application, starting with a CC++ specification.

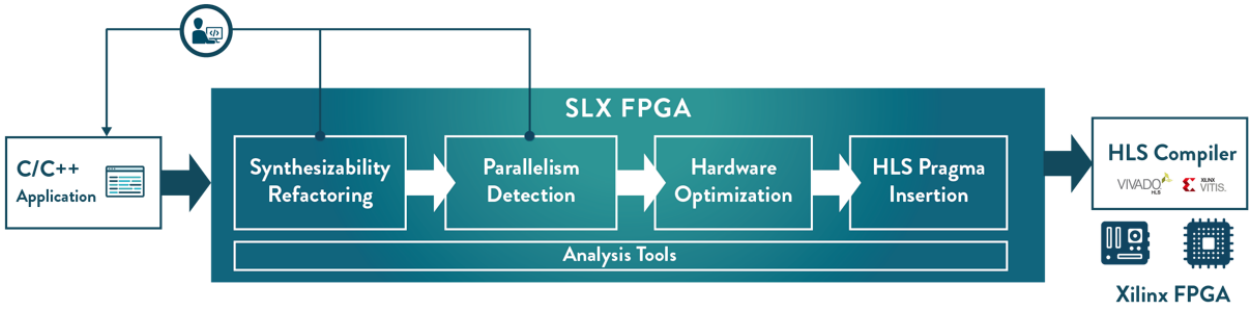

Fig. 1: SLX FPGA: automated workflow for HLS

SLX FPGA performs dynamic analysis to extract deep insight into the behavior of the code. Its analysis capabilities 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. SLX FPGA has been augmented to detect opportunities for performing loop interchange transformations. When it determines that a particular loop nest would result in better performance by interchanging loops, it inserts the necessary directives in the code to direct the SLX Plugin to perform the transformation. The tool flow is depicted in the figure below.

5.1 SLX FPGA Project

SLX FPGA version 2020.4 contains the SLX Plugin installed along with its other tool binaries and libraries. No additional setup steps are required to combine SLX FPGA and the SLX Plugin. Open the SLX FPGA

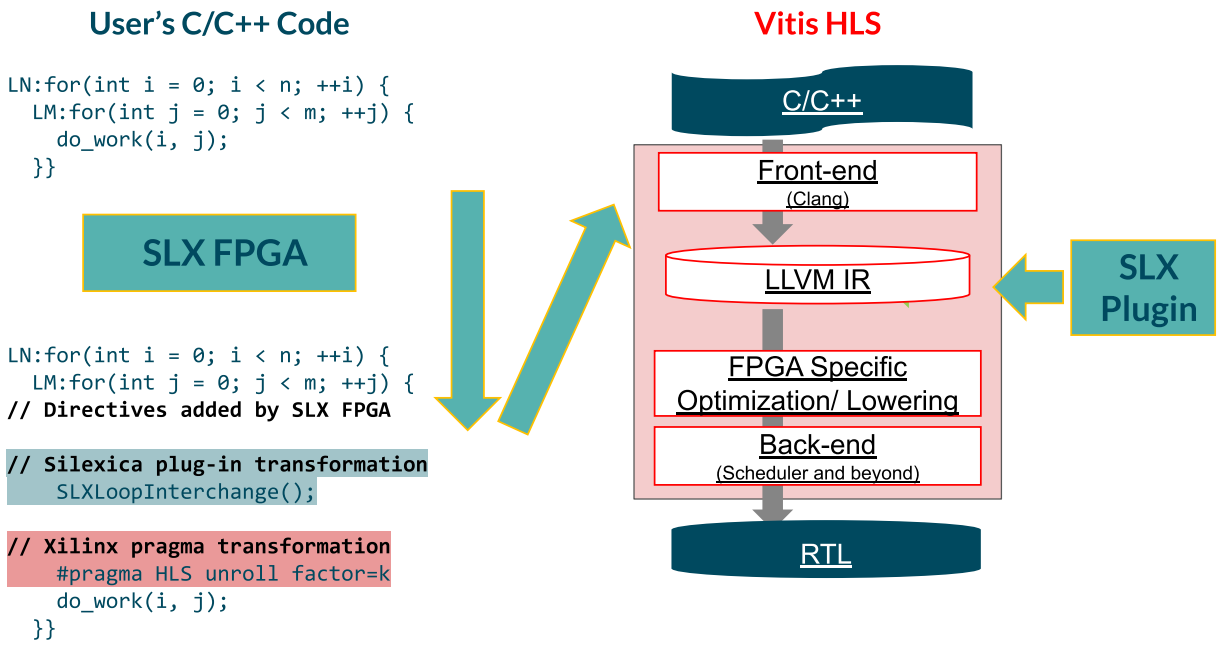

Fig. 2: SLX FPGA, the SLX Plugin and Vitis HLS used together to automatically determine and apply loop interchange directives.

GUI following the usual procedure and the tool will already be aware of the plugin (restricted to supported versions).

Attention: The SLX Plugin is only supported in SLX FPGA 2020.4 for Ubuntu 18.04 host machines.

The SLX FPGA GUI allows enabling and disabling the SLX Plugin and the corresponding loop interchange directives. When enabled, SLX FPGA will then take care of generating and adapting Tcl files, setting options, adding `_SLXLoopInterchange` to your source files, and optimizing your design.

For more information about SLX FPGA, refer to the SLX FPGA User Guide.

Chapter 6

Legal matter

6.1 Copyright notice and proprietary information

6.2 Destination control statement

All technical data contained in this publication is subject to the export control laws of the German Federal Republic. Disclosure to nationals of other countries contrary to Germany law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

6.3 Disclaimer

SILEXICA GMBH, MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

6.4 Trademarks

Silexica and certain Silexica product names are trademarks of Silexica GmbH. All other product or company names may be trademarks of their respective owners.

6.5 Third-Party links

Any links to third-party websites included in this document are for your convenience only. Silexica does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

6.6 Silexica GmbH

Lichtstr. 25
50825 Cologne, Germany
www.silexica.com