SILEXICA

APPLICATION NOTE

Using the SLX Plugin for Vitis HLS to optimize MRI Reconstruction

Introduction:

Magnetic Resonance Imaging (MRI) is one of the technologies that has enabled medical professionals to investigate, research, and diagnose complex issues within the human body. As would be expected, MRI is a complex technology that requires significant processing capabilities to convert the sampled raw data into meaningful images. As such, accelerators like GPUs and heterogeneous SoCs are used for MRI reconstruction. This application note outlines the performance improvement that can be achieved when using high-level synthesis and targeting the programmable logic element of a Xilinx heterogeneous SoC. This application note will also demonstrate how the SLX Plugin for Vitis HLS is used to improve the performance of an FPGA application by leveraging a new optimization pragma made available to developers for implementing a loop interchange transformation. The plugin guarantees a functionally correct loop interchange transformation for a pair of loops in a loop nest. Thus, not only does it relieve the developers of the hassle of always refactoring loops to explore optimization opportunities but also removes an important source of errors being injected during the optimization process.

contact@silexica.com
www.silexica.com

Send Feedback

What is the SLX Plugin?

With the release of Vitis 2020.2, Xilinx has opened elements of the LLVM Intermediate Representation layer of the Vitis HLS compiler to partners. Exploiting this capability, Silexica has developed the first SLX Plugin that provides developers with an opportunity to improve the latency and throughput of their FPGA applications. The plugin provides this increased performance by making a new pragma available for developers to use as they develop their HLS application.

The Silexica Plugin supplies the user with a single new pragma which provides the ability to perform a loop interchange transform. Loop interchange can be used with nested loops to effectively change the iteration order of the nested loops.

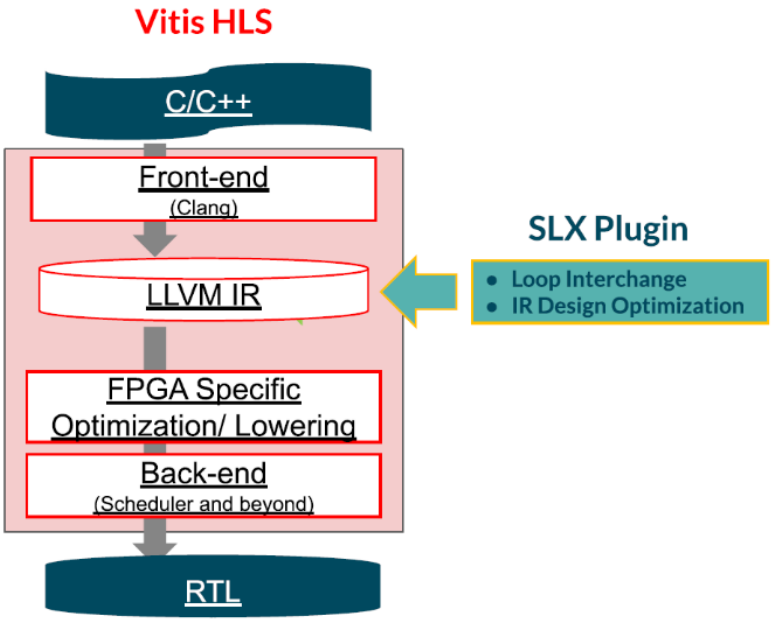

Figure 1: SLX Plugin overview

In this application note, we explain how to install the plugin and work through a detailed MRI reconstruction example demonstrating the capabilities of the new optimization pragma.

Installation

The following prerequisites are required in order to leverage the Silexica Plugin.

1. Ubuntu 18.04 Linux Operating System
2. Vitis HLS 2020.2

Provided the prerequisites are in place, the Silexica plugin can be downloaded from the Silexica website.

The download will be provided as a compressed tarball. We need to first extract the plugin files to begin installing the plugin. Once extracted, you will see a new directory with the version of the Silexica Plugin. Inside this directory, you will find several directories which include the user guide under the doc directory and example applications provided under the example directory, as shown in figure 2:

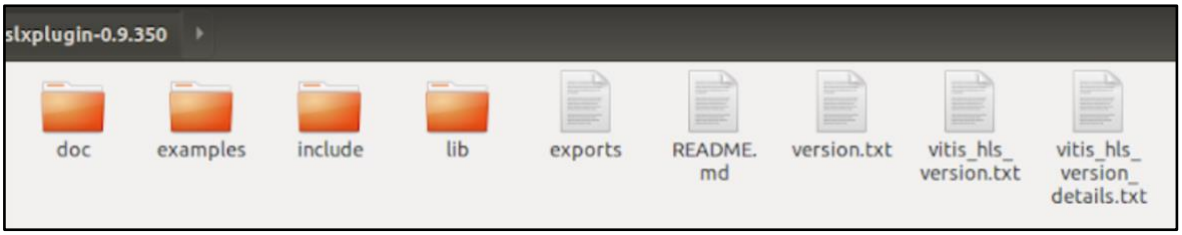

Figure 2: SLX plugin directory structure

Installing the plugin is very straight forward. Figure 3 highlights the installation steps. Using a terminal window, ensure you have run the Vitis HLS set-up script and then execute the command *source exports* within the Silexica Plugin directory. This will set the environment variable SLX_VITIS_PLUGIN which defines the location of the plugin so Vitis HLS can access it during compilation.

```
~$ source /tools/Xilinx/Vitis_HLS/2020.2/settings64.sh
~$ cd ~/Downloads/
~/Downloads$ tar -xzf slxplugin_download.tgz
~/Downloads$ source slxplugin-0.9.350/exports
```

Figure 3: SLX plugin installation steps

As the environment variable is only set within the current terminal session, this command must be re-entered to leverage the Silexica Plugin each time we start a new terminal. We can verify that the installation is correct by running one of the provided example designs. However, we should understand exactly how the loop interchange pragma works first.

Loop Interchange Pragma

At the simplest level, the loop interchange pragma works with nested loops to enable the iteration order of two variables to be changed. The simplest example of this is presented below where two nested loops are present.

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

}
```

In this example, the inner loop *(LoopM)* has the loop interchange pragma applied to it. This means it will be interchanged with its immediate parent, *LoopN*. Performing the loop interchange results in changing the iteration order of both variables from a row-major order to a column-major order. To further demonstrate how loop interchange works with a more complex situation, the previous example is expanded to include a third loop, moving the location of the pragma as presented in the example below.

```
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] );
                }

        }

}
```

In this example, the loop interchange pragma is applied to *LoopN* and its parent, *LoopO*. This means the normal iteration process of K then J then I becomes K then I then J. If the loop interchange pragma were to be positioned inside of *LoopM,* the order would, of course, be J then K then M.

Interchanging loops can assist the HLS compiler in relaxing conditions which limit parallelism or pipelining. For example, in the code example below there is a loop dependency which can prevent achieving an initiation interval of one clock cycle.

```
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];
        }

}
```

The bottleneck in the initiation interval arises in every increment of *LoopN*, where *LoopM* must be completed which means multiple accesses to the data memory are required. This bottleneck can be observed in the code because all three accesses are based on the *LoopN* iteration variable which is the outer variable.

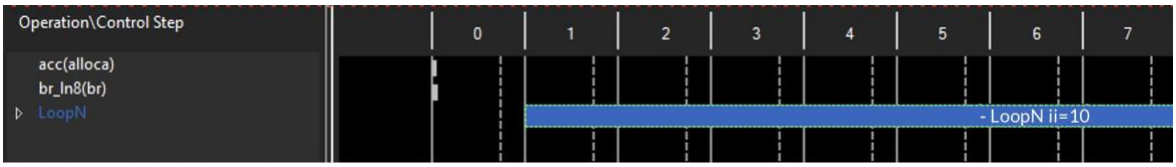

Figure 4: Schedule without loop interchange (ii=10)

Reconfiguring the loop using the loop interchange pragma ensures that the three iteration variables are controlled by the now inner loop. This enables the Vitis HLS compiler to optimize the design so that it can achieve an iteration interval of one, as shown in figure 5.

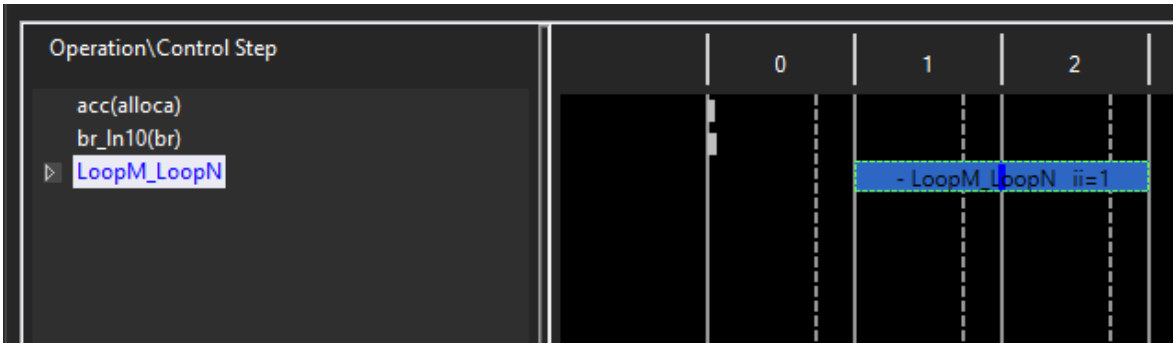

Figure 5: Schedule with loop interchange (ii=1)

MRI Reconstruction

Magnetic Resonance Imaging is used by medical professionals to observe the structure and function of biological tissue and is a very flexible diagnostic tool capable of imaging all areas of the body. Such detailed visibility provides significant information that can be used for both diagnosis and research. A typical MRI consists of two elements; the initial scan during which the data is acquired, followed by the reconstruction. During the scan, the data samples are captured along a predefined trajectory. These samples are special in nature and are in the k-space domain. Transforming the acquired samples into an understandable image occurs during the reconstruction phase. MRIs face the competing challenges of high-definition imaging, low signal to noise, and fast scan time. The complexity of the image reconstruction depends upon the sampling trajectory. A simple Cartesian scan direction provides the k-space samples aligned to a grid, allowing quick image reconstruction using a Fast Fourier Transform. A non-Cartesian scan, like a spiral trajectory, for example, will result in the k-space samples aligned in a more complex pattern, which requires advanced image reconstruction.

Due to the significant processing required for the reconstruction of an MRI image, it can take a considerable amount of time if a traditional CPU is used. It is not unusual for CPU processing to take several tens of minutes for image reconstruction. Of course, GPUs have been used to accelerate the image reconstruction time and to leverage the multiple GPU cores. One GPU-based algorithm for reconstruction is presented in the paper Accelerating Advanced MRI Reconstructions on GPU, published in the Journal of Parallel and Distributed Computing, 68(10), 1307–1318. This algorithm proposes three steps for image reconstruction:

1. Computing the data structure, Q, the convolution kernel.
2. Computing the vector $F^H d$, where F^H denotes the imaging process and d is the vector of data points.
3. Conjugate Gradient Linear Solver to find the image iteratively.

Given a reconstruction problem of N voxels and M scan data points, the computations of Q and $F^H d$ have complexity of O(MN) and O(N log N) respectively. Both algorithms use nested loops and are remarkably similar.

```
for (K = 0; k < numK; K_++)
        phiMag[K] = rPhi[K] * rPhi[K] + iPhi[K] * iPhi[K];


for (X = 0; X <numX; X++)
        for (K = 0; K < numK; K ++)
                exp = 2 * PI * (kx[K] * x[X] + ky[K] * y[X] +kz[K] * z[X];
                rQ[X] += phiMag[K] * cos(exp);
                iQ[X] += phiMag[K] * sin(exp)
```

Algorithm for calculating Q

```
for (K = 0; k < numK; K_++)
        rRho[K] = rPhi[K] *rD[K] + iPhi[K] * iD[K]
        iRho[K] = rPhi[K] *iD[K] - iPhi[K] * rD[K]

for (X = 0; X <numX; X++)
        for (K = 0; K < numK; K ++)
                exp = 2 * PI * (kx[K] * x[X] + ky[K] * y[X] +kz[K] * z[X];
                cArg = cos(exp)
                sArg = sin(exp)
                rFH[X] += rRho[K] * cArg - iRho[K] * sArg;
                iFH[X] += iRho[K] * cArg + rRho[K] * SArg;
```

Algorithm for calculating $F^H d$

To demonstrate the power of the loop interchange pragma, the Q algorithm was implemented in Vitis HLS and the Silexica Plugin was used to perform the loop interchange to demonstrate the increase in performance that can be achieved from its use.

The application consists of four files for the synthesizable implementation and test bench:

- mri_vhls.c – implementation of the Q algorithm for synthesis
- mri_vhls.h – header file for the Q algorithm implementation
- mri_vhls_tb.cpp – test bench for the Q algorithm
- result.h – pre-calculated results for the test bench which should match the generated data

The implementation of the Q algorithm for Vitis HLS is provided in listing 1.

```
14  // C code for the algorithm that compute Q - the
15  // first step of the advanced MRI reconstruction algorithm.
16  void mri_fk(float kphi_r[M], float kphi_i[M],
17              float kx[M],     float ky[M],     float kz[M],
18              float xx[8 * N], float xy[8 * N], float xz[8 * N],
19              float rQ[8 * N], float iQ[8 * N]) {
20
21    float PI_x_2 = (3.14 * 2.0);
22    float kphi_mag[M];
23    float exponent;
24
25  LOOP_MAGPHI:
26    for (int m = 0; m < M; ++m) {
27      kphi_mag[m] = kphi_r[m] * kphi_r[m] + kphi_i[m] * kphi_i[m];
28    }
29
30  LOOP_N:
31    for (int n = 0; n < 8 * N; ++n) {
32    LOOP_M:
33      for (int m = 0; m < M; ++m) {
34  _SLXLoopInterchange();
35        exponent = PI_x_2 * (kx[m] * xx[n] + ky[m] * xy[n] + kz[m] * xz[n]);
36        rQ[n] += kphi_mag[m] * cosf(exponent);
37        iQ[n] += kphi_mag[m] * sinf(exponent);
38      }
39    }
```

Listing 1: C implementation for the algorithm to compute Q

As can be seen from the source code, the loop interchange pragma is used on the nested loops. Performing a baseline implementation with the loop interchange pragma disabled, we see that the latency of both nested loops is considerable. Figure 6 shows the Vitis HLS schedule for this baseline implementation, with the overall latency of approx. 38 million clock cycles. The iteration latency for *Loop_N* is 9285 clock cycles, of which 9280 cycles are spent in the inner loop, *Loop_M,* which has an initiation interval of 9 clock cycles.

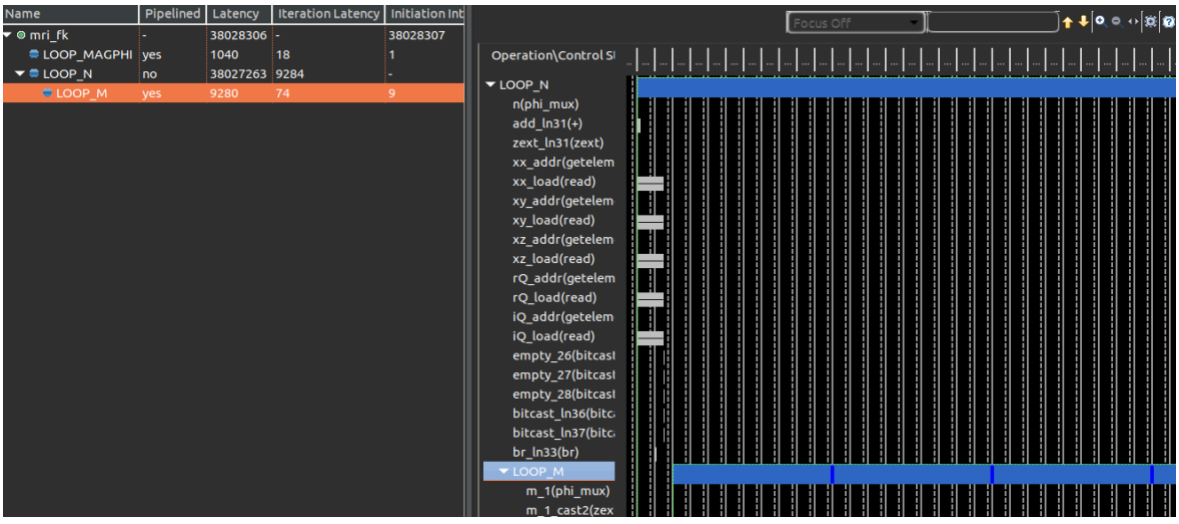

Figure 6: Vitis HLS Schedule for MRI reconstruction example without loop interchange

Figure 7 shows the part of the detailed schedule for *Loop_M***.** The most prominent bottleneck for this loop is identified by the II violation due to a feedback path (the green arrow in figure 7). The root cause of this feedback path can be traced to the reduction operations on lines 36 and 37 of the source code. Since the reduction operations depend on the induction variable of the outer loop, all iterations of the inner loop have to read and update the same elements of the rQ and iQ arrays thus creating a loop carried dependency.

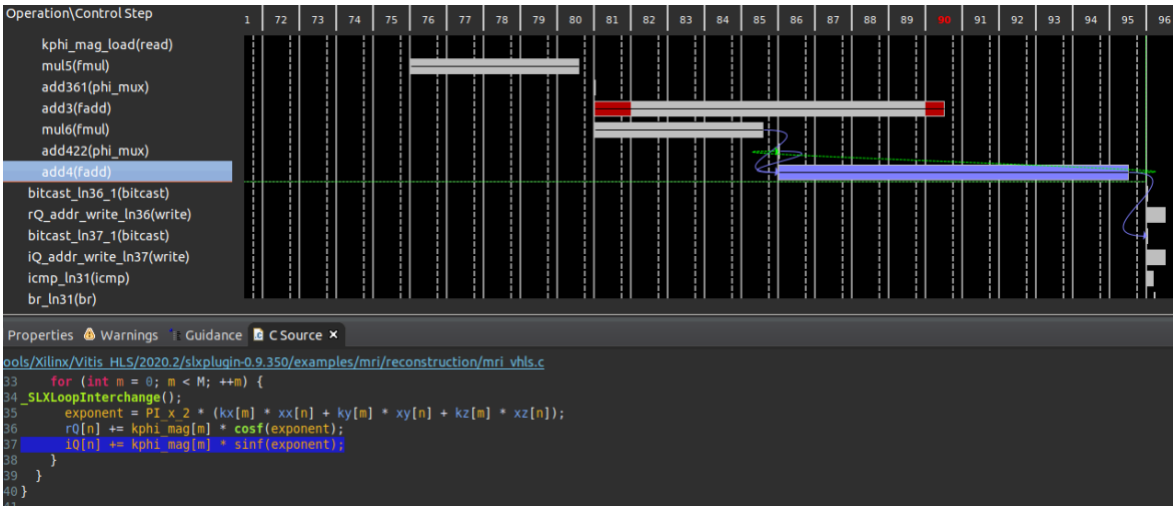

Figure 7: Detailed schedule for the inner loop showing a feedback path due to an LCD

Enabling the loop interchange pragma interchanges *Loop_N* and *Loop_M*. The iterations of the inner loop now read and update different elements of the rQ and iQ arrays, thus removing the feedback path, as shown by the schedule in figure 8. This enables the inner loop to be pipelined with an initiation interval of one.

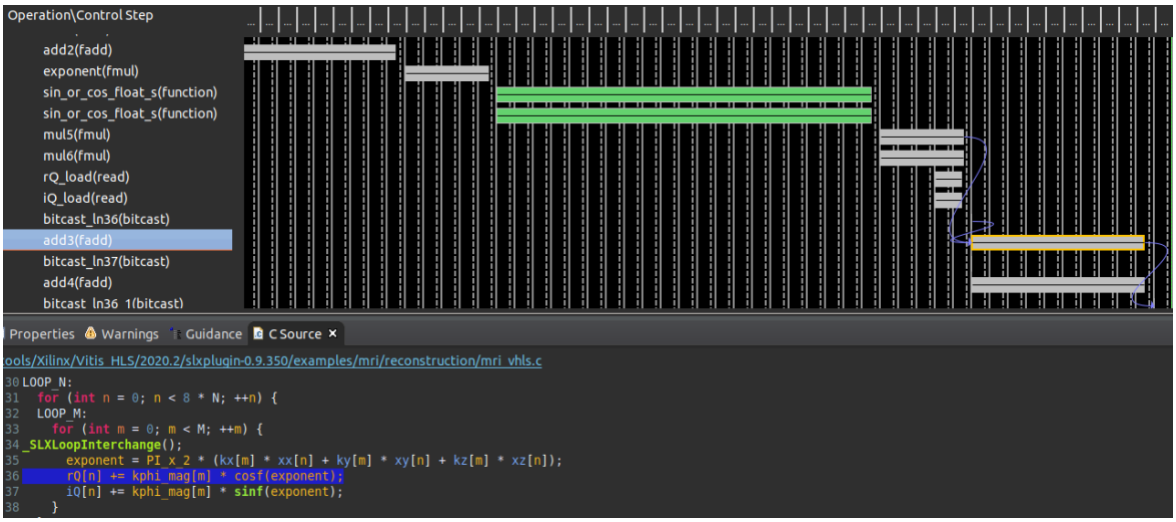

Figure 8: Schedule for the inner loop after loop interchange

Table 1 shows a comparison of latencies of the different loops with and without the SLXLoopInterchange pragma enabled. Interchanging the loop reduces the initiation interval of the inner loop from 9 cycles to 1. Whereas, the latency of the outer loop is reduced from 38 million to 4.2 million.

	Inner loop II (cycles)	Latency (cycles)
Baseline	9	38027263
Loop Interchange enabled	1	4267007

Table 1: Performance results with and without the SLX Loop Interchange pragma

Wrap Up

Reconstructing MRI images requires a complex algorithm that presents several bottlenecks that can limit performance. Accelerating the reconstruction process using programmable logic enables the inherent parallel nature of logic to be exploited to increase performance of the image reconstruction. However, the algorithm performance when implemented in logic is constrained by the loop carried dependencies. Leveraging the loop interchange pragma enables this bottleneck to be removed and the performance to be significantly increased. The implementation of the Q algorithms nested loop takes only 11% of the original latency when the loop interchange pragma is applied. This translates to an 8.9x speedup and represents a significant performance boost, enabling faster image reconstruction. Most importantly, it provides developers with the opportunity to implement a functionally correct loop transformation with a simple pragma without going through the often cumbersome and error-prone process of refactoring nested loops. It is important to note that SLX FPGA will automatically insert the pragma, using heuristics to determine if a loop interchange transformation will provide a benefit.

Silexica Europe:

Silexica GmbH
Lichtstr. 25
50825 Köln
Germany

+49 221 986 5619 0
europe@silexica.com

Silexica USA:

Silexica Inc.
2033 Gateway Place,
San Jose, CA 95110
USA

+1 469-613-5049
usa@silexica.com

Silexica Japan:

Silexica Japan KK
8F, Shinyokohama 2-3-3,
Kohoku-ku Yokohama-shi,
Kanagawa, 222-0033
Japan

+81-80-5655-9777

japan@silexica.com

About Us:

Silexica enables a software-defined world composed of intelligent products. We disrupt the developer's journey from software to application-specific hardware systems, democratizing accelerated computing to help build a smart, connected and safe world.

Our mission is to provide software development tools reducing time-to-market of innovative software IP and intelligent products. Enabled by deep software analysis, heterogeneous hardware awareness and quick design space exploration, the SLX programming tools significantly accelerate the journey from software to application-specific hardware systems, empowering our customers to win markets.

Founded in 2014, Silexica is headquartered in Germany with offices in the US and Japan. It serves innovative companies in the automotive, robotics, wireless communications, aerospace, and financial industries and has received $28M in funding from international investors.

For more infomation, please visit our website or follow us on Linkedin, Twitter, Facebook and Youtube.

Silexica - Democratizing Accelerated Computing