T2SP Programming Guide

Hongbo Rong

April 19, 2022

Contents

1 Overview **3**

2 Rewriting math equations into basic UREs **5**
- 2.1 Step 1: Iterative form . . . 5
- 2.2 Step 2: Recursive form . . . 6
- 2.3 Step 3: DSA (Dynamic Single Assignment) . . . 6
- 2.4 Step 4: Full index form . . . 6
- 2.5 Step 5: UREs . . . 6
  - 2.5.1 First way to translate AREs into UREs: drawing a dataflow graph . . . 7
    - 2.5.1.1 Expressing a pipeline . . . 7
  - 2.5.2 Second way to translate AREs into UREs: Calculating a propagation direction . 7
  - 2.5.3 Testing correctness of UREs in standard C . . . 8
  - 2.5.4 Expressing the UREs in T2S . . . 9

3 Writing tiled UREs **11**

4 Multi-threading **12**

5 Building a systolic array **13**

6 How to compile and run **14**

Listings

2.1 Testing the correctness of the UREs in Table 2.1 in standard C code. 9
2.2 Testing the correctness of the UREs in Table 2.1 in T2S. 9

Chapter 1

Overview

T2SP generates accelerators for dense tensor computes, and a general workflow is shown in Fig. 1.1. Usually, such a compute is described as one or a few math equations. We can rewrite these equations to be recursive, the so-called Uniform Recursive Equations or UREs. With UREs, we can accurately control input and output data to flow in a pipeline fashion during the compute.

We can write basic UREs without tiling the compute domain. Then we tile the compute domain for data locality and reuse. Correspondingly, we need translate the basic UREs into final UREs after tiling, which is straightforward.

UREs can be expressed in C/C++. After adding some helper code necessary for testing, we can compile them in a standard C/C++ compiler, and check if the UREs are correct vs. the original math equations.

The compute domain is iterated with a loop nest. If the compute is to run on a GPU, we need decide which loops are block loops, and inside a block, which loops are thread loops: An iteration of the thread loops will be turned into a thread. If the compute is to run on an FPGA, however, we always use a single thread for the entire compute, and thus there is no block or thread loop.

Inside a thread, we will build a systolic array in a space-time transform. So we need decide which loops are space loops.

Finally, we add an I/O network for the systolic array. The I/O network is composed of multiple memory levels. The input/output data are loaded/unloaded through the I/O network into/out of the systolic array. Once in the systolic array, the data flow through the array according to the UREs.

Now for the original compute, we have a complete T2SP program, or more accurately, a specification, since such a program only specifies what to implement (e.g. space-time transform and I/O network), but leave the actual implementation to the T2SP compiler. In other words, the program controls the compiler to generate the expected accelerator.

For an FPGA, the T2SP compiler generates OpenCL device code and a C interface, compile and run these generated code by invoking the downstream FPGA tools. The programmer can use the emulator tool to verify correctness of the generated code, and use the synthesis tool to produce a bitstream. The programmer can write CPU host code and call the device code through the C interface, just like calling any normal C function on a CPU. That call would offload the compute to the emulator or an actual FPGA to run.

For a GPU, the T2SP compiler generates CM device code, and invoke the CM compiler to build a binary. The programmer need write CPU host code that offloads the device code to a GPU.

A beginner might encounter several hurdles in using T2SP: (1) how to write UREs? (2) how to tile loops? (3) how to decide block, thread, and space loops? and (4) how to design I/O? There are numerous valid answers for each of these questions. Fortunately, we do not have to be mathematicians, algorithm experts, or computer architects in order to effectively address the questions. Based on intuitive rules, any programmer may master some simple programming skills quickly, and use the skills to accelerate real-world workloads.

We will describe the steps in more details below.

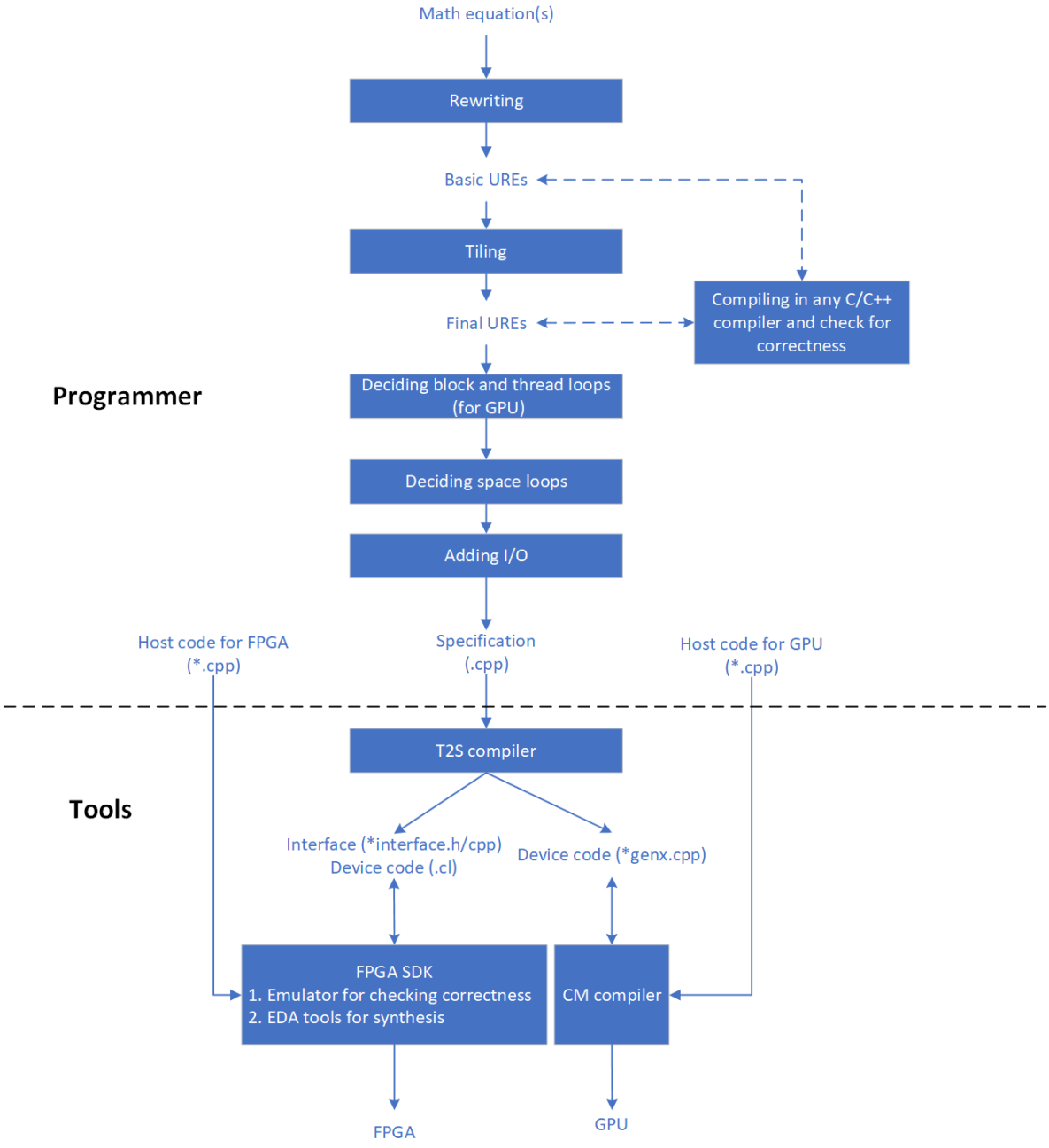

Figure 1.1: Overall flow

Chapter 2

Rewriting math equations into basic UREs

Based on the work of Gusev and Evans [1, 2], we can translate math equations into UREs, following the simple steps below. Going through an example, we would find that this process is pretty much mechanical and intuitive, and only middle-school level of math knowledge is needed. So do not be scared by the math symbols. In fact, you want to work out the UREs through the simple math process, since the correctness can be proved and that would save you time in debugging.

We strongly recommend a beginner tries one example so as to quickly master the skills. These skills are generally applicable to real world problems.

Use auto-regressive filter [3] for example:

$$X_t = c + \sum_{i=1}^{I} \varphi_i X_{t-i} + \varepsilon_t \tag{2.1}$$

where c is a constant, $\varphi_1, ...\varphi_I$ are the parameters, ε_t is white noise, and X_t is the output at the current time t and is dependent on the previous I outputs $X_{t-1}, ..., X_{t-I}$.

Table 2.1 shows a complete process how UREs are derived for the problem, and the steps are explained in more detail below:

Step	Equations		Initial values	
1: Iterative form	$X_t = c + \sum_{i=1}^{I} \varphi_i X_{t-i} + \varepsilon_t$	$t = 1, ..., T$	$X_{t-i} = 0$	if $t - i \leq 0$
2: Recursive form	$X_t = X_t + \varphi_i X_{t-i}$	$t = 1, ..., T$ $i = 1, .., I$	$X_t = c + \varepsilon_t$ $X_{t-i} = 0$	if $t - i \leq 0$
3: DSA	$X_t^i = X_t^{i-1} + \varphi_i X_{t-i}^I$	$t = 1, ..., T$ $i = 1, .., I$	$X_t^0 = c + \varepsilon_t$ $X_{t-i}^I = 0$	if $t - i \leq 0$
4: Full index form	$X_t^i = X_t^{i-1} + \varphi_i^0 X_{t-i}^I$	$t = 1, ..., T$ $i = 1, .., I$	$X_t^0 = c + \varepsilon_t^0$ $X_{t-i}^I = 0$	if $t - i \leq 0$
5: UREs	$\Phi_t^i = (t-1=0)?\varphi_i^0 : \Phi_{t-1}^i$ $\chi_t^i = (t-1=0)?0 : (i-1=0?X_{t-1}^{i-(1-I)} : \chi_{t-1}^{i-1})$ $X_t^i = (i-1=0?c+\varepsilon_t^0 : X_t^{i-1}) + \Phi_t^i \chi_t^i$	$t = 1, ..., T$ $i = 1, .., I$		

Table 2.1: Deriving UREs for auto-regressive filter. Here $c?a : b$ is an expression returning a when condition c is true, and b otherwise.

2.1 Step 1: Iterative form

First, write down the math equation(s) of the original problem. Usually, in an equation, a domain is iterated (like $i = 1, ..., I$ and $t = 1, ..., T$), and some variable (like X) is computed. A variable might

have some initial values (like $X_{t-i} = 0$ for $t - i \leq 0$).

2.2 Step 2: Recursive form

Translating the iterative form to a recursive form is straightforward: according to the iterative form, initialize a variable (e.g. $X_t = c + \varepsilon_t$), and update the variable every iteration with a new value based on its previous value (in the form of $X_t = X_t + ...$).

2.3 Step 3: DSA (Dynamic Single Assignment)

When updating a variable every iteration, save it to a distinct memory location. Every reference to a variable thus exposes the iteration in which the variable is defined. For example, $X_t = X_t + ...X_{t-i}$ is changed into $X_t^i = X_t^{i-1} + ...X_{t-i}^I$. After this renaming, it is clear that the 3 references to X are referring to the X values defined in the current iteration $\binom{i}{t}$ and previous iterations $\binom{i-1}{t}$ and $\binom{I}{t-i}$, respectively. Consequently, dataflow/dependences between these iterations are made explicit.

For another example, the initialization $X_t = c + \varepsilon_t$ is changed into $X_t^0 = c + \varepsilon_t$ by adding one 0 for the missing index i. X_t^0 refers to the X value defined in iteration $\binom{0}{t}$, which is outside the domain, because index i starts from 1 with a step of 1. In general, if index i starts from s with a step h, the initial value should be X_t^{s-h}.

2.4 Step 4: Full index form

Now variables are referenced with full indices, but constants are not: φ_i and ε_t are inputs never modified when computing X_t (The other constant c is just a number and we do not care). Similar to the handling of initialization in step 3, we give these constants full indices by adding 0's for the missing index i: change φ_i and ε_t into φ_i^0 and ε_t^0. They refer to the φ and ε values defined in iteration $\binom{0}{i}$ and $\binom{0}{t}$, respectively, which are outside the domain. In general, if iteration index i starts from s with a step h, we should rename φ_i and ε_t into φ_i^{s-h} and ε_t^{s-h}.

After being full indexed, variables and constants will be processed in the same way.

At this point, the equations we get are AREs (Affine Recurrence Equations), that is, the current iteration $\binom{i}{t}$ reads a value defined in a previous iteration with a distance d that is in the form of $d = A\binom{i}{t} + d_0$, where A is a matrix and d_0 a constant vector. In other words, there is a read-after-write affine dependence between the two iterations. The dependence distance can be calculated as the current iteration - the previous iteration = the write's index - the read's index: Remember that every write/read is fully indexed with the iteration that defines its value; thus the write is indexed with the current iteration, the read is indexed with the previous iteration. See Table 2.2 for all the dependences:

Dependence No.	Write	Read	Dependence distance
1	X_t^i	X_t^{i-1}	$\binom{i}{t} - \binom{i-1}{t} = \binom{1}{0}$
2	X_t^i	φ_i^0	$\binom{i}{t} - \binom{0}{i} = \binom{i}{t-i} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix}\binom{i}{t}$
3	X_t^i	X_{t-i}^I	$\binom{i}{t} - \binom{I}{t-i} = \binom{i-I}{i} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}\binom{i}{t} + \binom{-I}{0}$

Table 2.2: Dependences of the full index form in step 4 of Table 2.1

.

2.5 Step 5: UREs

We translate AREs into UREs by converting a broadcast into a pipeline. After that, every dependence has a constant distance uniformly in the entire domain.

There are two ways to convert a broadcast into a pipeline, either graphically or mathematically.

2.5.1 First way to translate AREs into UREs: drawing a dataflow graph

We can draw the dataflow and intuitively figure out how to change a broadcast into a pipeline, as exemplified in Fig. 2.1. According to the 2nd dependence in Table 2.2, originally, a datum φ_i^0 is broadcast to iterations X_t^i for all t, as shown in the left of the figure.

Equivalently, the same datum can be loaded in an iteration at a boundary of the domain, and from that iteration, propagated in a pipeline fashion to all the other iterations, as shown in the right of the figure. As we can see, φ_i^0 is loaded at a bounary iteration $(i, 1)$, and then is propagated to iteration $(i, 2)$, and from there to iteration $(i, 3)$, etc.

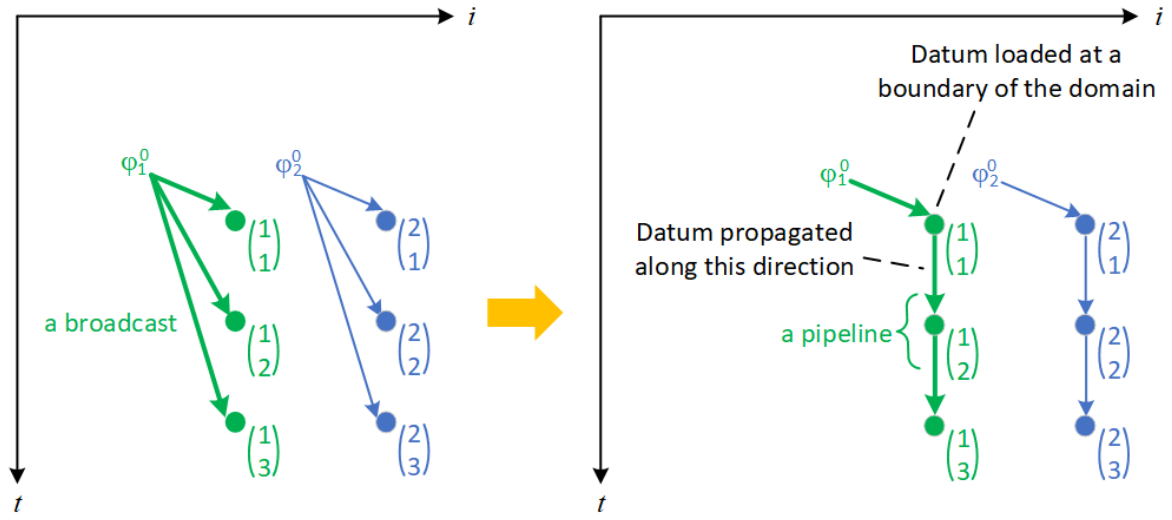

Figure 2.1: For the 2nd dependence of auto-regressive filter in Tabel 2.2, the broadcasts due to this dependence are changed to pipelines. Here we assume $T = 3$, and every point is an iteration annotated with its indices $\binom{i}{t}$.

2.5.1.1 Expressing a pipeline

Now we can modify the full index form

$$X_t^i = ...\varphi_i^0... \tag{2.2}$$

into

$$X_t^i = ...\Phi_t^i... \tag{2.3}$$

where Φ values are propagated along the t dimension until out of the domain:

$$\Phi_t^i = (t - 1 = 0)?\varphi_i^0 : \Phi_{t-1}^i \tag{2.4}$$

As you can see, the keys to change a broadcast into a pipeline are: (1) the direction of the pipeline, along which data would be propagated from one iteration to the next, and (2) the boundary conditions when the pipeline is fed with some initial values.

In the same way, we can convert a broadcast due to the third dependence into a pipeline (Could you do it?). After that, we get the UREs shown in step 5 of Table 2.1.

2.5.2 Second way to translate AREs into UREs: Calculating a propagation direction

We can generalize the example in Fig. 2.1, and directly find out a propagation direction vector in simple math:

If a read-after-write dependence is affine, i.e. the distance d is in the form of $Az + d_0$, where A is a matrix, z is the current iteration, and d_0 is a constant vector, then the dependence incurs broadcasts of values, and such a broadcast can be changed into a pipeline by propagating a value along a direction r that is a solution of $(E - A)r = \mathbf{0}$, where E is the identity matrix.

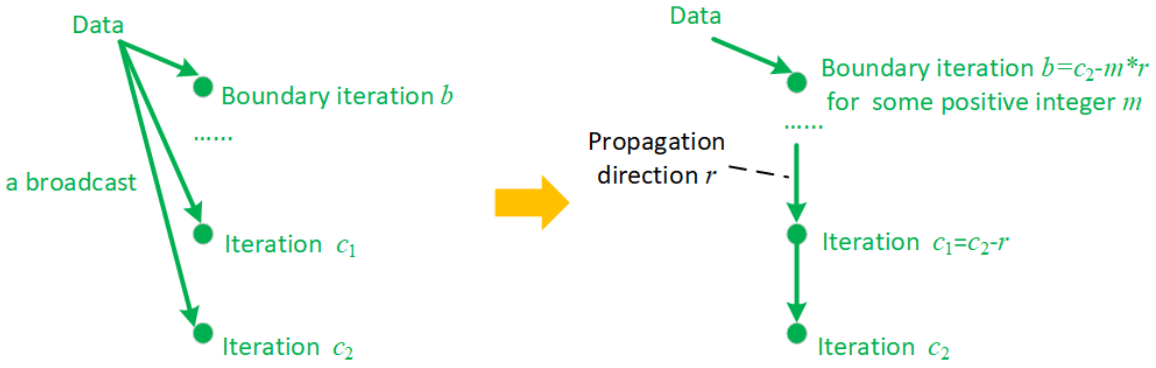

Figure 2.2: Changing a broadcast into a pipeline

The left part of Fig. 2.2 shows that due to an affine dependence, some data are broadcast to multiple consumers, including an iteration b at the boundary of the domain, and two other iterations c_1 and c_2, etc. Remember that data are fully indexed, i.e. they can be considered having been defined in an iteration y, even though that iteration is actually out of the domain.

Let the dependence distance be $d = Az + d_0$. Because both iteration $c1$ and c_2 get data from the same iteration y, using the dependence distance, we have

$$y = c_2 - (Ac_2 + d_0) = c_1 - (Ac_1 + d_0) \tag{2.5}$$

So

$$(E - A)c_2 = (E - A)c_1 \tag{2.6}$$

Let $r = c_2 - c1$, we have

$$(E - A)r = 0. \tag{2.7}$$

Therefore, we can imagine that a datum that is defined outside the domain is first sent to a boundary iteration b, and then following a propagation direction r to the next iteration $b + r$, and so on, and eventually the data reach iteration c_1, and then c_2, etc. This constructs a pipeline as shown in the right part of Fig. 2.2.

For example, the second dependence for the auto-regressive filter is $\left(\begin{smallmatrix}1 & 0\\ -1 & 1\end{smallmatrix}\right)\left(\begin{smallmatrix}i\\ t\end{smallmatrix}\right)$ (Table 2.2), and thus $A = \left(\begin{smallmatrix}1 & 0\\ -1 & 1\end{smallmatrix}\right)$. Solve the equation $(E - A)r = \left(\begin{smallmatrix}0 & 0\\ 1 & 0\end{smallmatrix}\right)r = \mathbf{0}$. We get a solution $r = \left(\begin{smallmatrix}0\\ *\end{smallmatrix}\right)$, where the second element (the t dimension) can be arbitrary. Take a non-zero solution $r = \left(\begin{smallmatrix}0\\ 1\end{smallmatrix}\right)$, and we see exactly from the right part of Fig. 2.1 that a pipeline can be constructed along the t dimension for a datum.

With this propagation direction, we convert a broadcast due to the second dependence into a pipeline in the same way as shown in Section 2.5.1.1. Following the same approach, for the third dependence, we can find a propagation direction $r = \left(\begin{smallmatrix}1\\ 1\end{smallmatrix}\right)$, and convert the broadcasts due to this dependence into pipelines as well (Could you do it?). After that, we get the UREs shown in step 5 of Table 2.1.

2.5.3 Testing correctness of UREs in standard C

We can express the UREs in Table 2.1 in standard C, adding necessary helping code for testing, and see if the UREs can produce correct results. See Listing 2.1.

Listing 2.1: Testing the correctness of the UREs in Table 2.1 in standard C code.

```
1  // auto-regressive-filter-testing-ures-in-c.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  int main() {
7      const int T = 10;
8      const int I = 3;
9      const int c = 2;
10
11     // Inputs.
12     int phi[1][I+1], epsilon[1][T+1];
13     for (int i = 1; i <= I; i++) { phi[0][i] = rand(); }
14     for (int t = 1; t <= T; t++) { epsilon[0][t] = rand(); }
15
16     // Arrays storing the values of UREs.
17     int Phi[I+1][T+1];
18     int Chi[I+1][T+1];
19     int X[I+1][T+1];
20
21     // Run the UREs.
22     for (int t= 1; t <= T; t++) {
23       for (int i = 1; i <= I; i++) {
24         Phi[i][t] = (t-1 == 0) ? phi[0][i] : Phi[i][t-1];
25         Chi[i][t] = (t-1 == 0) ? 0 : (i-1 == 0 ? X[i-(1-I)][t-1] : Chi[i-1][t-1]);
26         X[i][t]   = (i-1 == 0 ? c+epsilon[0][t] : X[i-1][t]) + Phi[i][t]*Chi[i][t];
27       }
28     }
29
30     // Validate correctness
31     int golden[T+1];
32     for (int t= 1; t <= T; t++) {
33       golden[t] = c + epsilon[0][t];
34       for (int i = 1; i <= I; i++) {
35         if (t - i > 0) {
36            golden[t] += phi[0][i] * golden[t - i];
37         }
38       }
39       assert(golden[t] == X[I][t]);
40     }
41     printf("Success!\n");
42 }
```

Now test it:

```
$gcc auto-regressive-filter-testing-ures.c
$./a.out
Success!
```

2.5.4 Expressing the UREs in T2S

It is straightforward to express the UREs in T2S. See Listing 2.2. The code is similar to the C code before. The major difference is that the T2S code builds a symbolic dataflow graph where inputs are symbolic parameters and computation is symbolic expressions (Line 13-27), then instantiates the graph to compute with concrete inputs (Line 29-37), and after that compares with a golden baseline for correctness.

Listing 2.2: Testing the correctness of the UREs in Table 2.1 in T2S.

```
1  // auto-regressive-filter-testing-ures-in-t2s.cpp
2
3  // The only header file to include
4  #include "Halide.h"
5  using namespace Halide;
6
7  int main()
```

```
8  {
9      const int T = 10;
10     const int I = 3;
11     const int c = 2;
12
13     /**** Build a dataflow graph for computing the auto regressive filter. ****/
14
15     // Inputs are 2-dimensional arrarys.
16     ImageParam phi(Int(32), 2), epsilon(Int(32), 2);
17
18     // UREs
19     Var  i("i"), t("t");
20     Func Phi("Phi",Int(32),{i,t}), Chi("Chi",Int(32),{i,t}), X("X", Int(32), {i,t});
21     Phi(i, t) = select(t-1 == 0, phi(0, i), Phi(i, t-1));
22     Chi(i, t) = select(t-1 == 0, 0, select(i-1==0, X(i-(1-I), t-1), Chi(i-1, t-1)));
23     X(i, t)   = (select(i-1 == 0, c+epsilon(0,t), X(i-1,t))) + Phi(i,t) * Chi(i, t);
24
25     // Put all the UREs inside the same loop nest, and explicitly set the loop
           bounds.
26     Phi.merge_ures(Chi, X)
27        .set_bounds(i, 1, I, t, 1, T);
28
29     /**** Instantiate the dataflow graph ****/
30
31     // Set inputs and run the dataflow graph on a CPU to compute the output.
32     Buffer<int> _phi(1, I+1), _epsilon(1, T+1), _X(I+1, T+1);
33     for (int i = 1; i <= I; i++) { _phi(0, i) = rand(); }
34     for (int t = 1; t <= T; t++) { _epsilon(0, t) = rand(); }
35     phi.set(_phi);
36     epsilon.set(_epsilon);
37     _X = X.realize(get_host_target()/*Get a CPU*/);
38
39     /**** Validate correctness ****/
40
41     Buffer<int> golden(T+1);
42     for (int t= 1; t <= T; t++) {
43       golden(t) = c + _epsilon(0, t);
44       for (int i = 1; i <= I; i++) {
45         if (t - i > 0) {
46            golden(t) += _phi(0, i) * golden(t - i);
47         }
48       }
49       assert(golden(t) == _X(I, t));
50     }
51     printf("Success!\n");
52     return 0;
53 }
```

Now test it:

```
$ export T2S_PATH=path_to_your_t2s_installation
$ export PATH=$T2S_PATH/Halide/bin:$T2S_PATH/install/gcc-7.5.0/bin:$PATH
$ export LD_LIBRARY_PATH=$T2S_PATH/Halide/bin:$LD_LIBRARY_PATH
$ g++ -I$T2S_PATH/Halide/include -L$T2S_PATH/Halide/bin -lHalide  -std=c++11 \
      auto-regressive-filter-testing-ures-in-t2s.cpp
$ ./a.out
Success!
```

Chapter 3

Writing tiled UREs

TBD.

Chapter 4

Multi-threading

TBD.

Chapter 5

Building a systolic array

TBD.

Chapter 6

How to compile and run

Suppose a specification file, named `a.cpp`, is ready, and it contains such statements:

```
Func SomeKernel(Place::Device);
definition of SomeKernel
SomeKernel.compile_to_host("interface", parameters of SomeKernel, IntelFPGA);
```

And suppose a host file `host.cpp` is ready to call `SomeKernel`.

1. **Set up environment**.

   Under the T2SP root directory:

   ```
   source setenv.sh (local | devcloud) fpga
   ```

2. **Compile the specification**.

   ```
   # If you intend to emulate your design on a CPU:
   g++ a.cpp $COMMON_OPTIONS_COMPILING_SPEC $EMULATOR_LIBHALIDE_TO_LINK

   # Or if you intend to execute it on real FPGA hardware:
   g++ a.cpp $COMMON_OPTIONS_COMPILING_SPEC $HW_LIBHALIDE_TO_LINK
   ```

3. **Run the specification to generate the kernel**.

   ```
   # For emulation:
   env BITSTREAM=a.aocx AOC_OPTION="$COMMON_AOC_OPTION_FOR_EMULATION" ./a.out

   # Or for execution:
   env BITSTREAM=a.aocx AOC_OPTION="$COMMON_AOC_OPTION_FOR_EXECUTION" ./a.out
   ```

   For the kernel, this command generates an OpenCL file (named `a.cl` after name of the environment variable `BITSTREAM`), invokes Intel FPGA SDK for OpenCL to synthesize the OpenCL file into a device bitstream (`a.aocx`), and generates `interface.h/cpp` for the host to invoke the kernel, as if the kernel is a common CPU function.

   You may modify the generated OpenCL file manually. A common debugging practice is to add `printf` statements inside the generated OpenCL file, e.g. inserting `printf("Variable v=%f", v)` to dump the value of variable `a`. Note `printf` works only with emulation.

   After any modification, you may re-generate the bitstream in the following commands:

   ```
   # For emulation:
   aoc $COMMON_AOC_OPTION_FOR_EMULATION a.cl -o a.aocx

   # Or for execution:
   rm -rf a.aoc* a/
   aoc $COMMON_AOC_OPTION_FOR_EXECUTION a.cl -o a.aocx
   ```

4. **Compile the host file**

```
# For emulation:
g++ host.cpp interface.cpp $COMMON_OPTIONS_COMPILING_HOST_FOR_EMULATION -o host.out

# Or for execution:
g++ host.cpp interface.cpp $COMMON_OPTIONS_COMPILING_HOST_FOR_EXECUTION -o host.out
```

   When compiling the host file, the kernel will be linked to the host code through the interface.

5. **Run the host file**

```
# For emulation:
env BITSTREAM=a.aocx CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1 \
    INTEL_FPGA_OCL_PLATFORM_NAME="$EMULATOR_PLATFORM" ./host.out

# For execution:
# DevCloud A10PAC FPGA only: convert the signed bitstream to be unsigned first
#   Type `y` when prompted
    source $AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh \
           -H openssl_manager -i a.aocx -r NULL -k NULL -o a_unsigned.aocx
    mv a_unsigned.aocx a.aocx
# Offload and run
aocl program acl0 a.aocx
env BITSTREAM=a.aocx INTEL_FPGA_OCL_PLATFORM_NAME="$HW_PLATFORM" ./host.out
```

6. **Debug a kernel (in an emulator only)**

   Just like debugging any executable in `gdb`, you would like to add `gdb` right before `./host.out`:

```
env BITSTREAM=a.aocx CL_CONTEXT_EMULATOR_DEVICE_INTELFPGA=1 \
    INTEL_FPGA_OCL_PLATFORM_NAME="$EMULATOR_PLATFORM" gdb ./host.out

# Now inside the gdb, set a breakpoint at the kernel
(gdb) b SomeKernel

# Run. Gdb will stop at the kernel's entrance.
(gdb) r
```

   Note: whether `gdb` could successfully stop at the specified kernel seems to depend on the version of `aoc` (i.e. Intel FPGA SDK for OpenCL). We verified that the debugger stops at a kernel with aoc 19.2.0 on an S10 FPGA, but somehow does not stop with aoc 19.4.0 on an A10 FPGA.

Note: if you want to re-run the above commands, remove the previously generated bitstream and intermediate files:

```
rm -rf a.* a/ *interface.* *.out exec_time.txt
```

Bibliography

[1] Marjan Gusev and David J. Evans. Algorithm transformations for the data broadcast elimination method. Technical Report 646, Parallel Algorithm Research Centre, Loughborough University of Technology, Loughborough, Leicestershire LE11 3TU, October 1991.

[2] Marjan Gusev and David J. Evans. Elimination of the computational broadcast : An application to the qr decomposition algorithm. Technical Report 676, Parallel Algorithm Research Centre, Loughborough University of Technology, Loughborough, Leicestershire LE11 3TU, January 1992.

[3] WikiPedia. Autoregressive model. https://en.wikipedia.org/wiki/Autoregressive_model#Definition.