DATE'24 Tutorial – CGRA-ME 2.0
Lab Manual

Omar Ragheb, Stephen Wicklund, Kentaro Sano[†], Jason Anderson,
Dept. of Electrical and Computer Engineering, University of Toronto, Canada
[†]RIKEN Center for Computational Science (R-CCS), Japan

2024-03-26

Introduction

CGRA-ME is an open-source CGRA modeling and exploration framework actively being developed at the University of Toronto. CGRA-ME is intended to facilitate research on new CGRA architectures and new CAD algorithms for CGRAs.

This lab's goals are to allow participants to explore the open-source CGRA-ME 2.0 platform [4] by walking through the processes of mapping applications onto an architecture, modifying an architecture, generating RTL for the architecture, and performing a functional RTL simulation. We will mainly be targeting the RIKEN CGRA, as discussed in the lecture part of this tutorial.

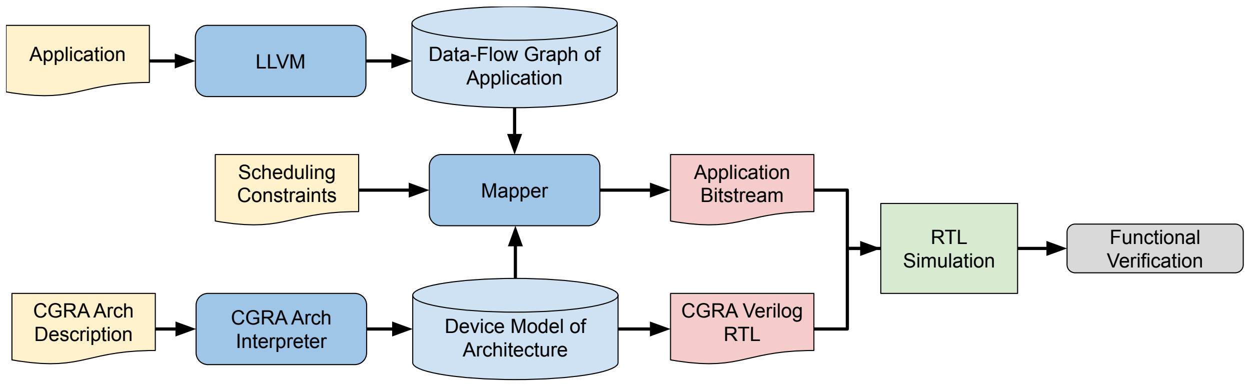

Figure 1: An overview of the CGRA-ME flow.

Set-Up

To follow along with this lab manual, you will need to have CGRA-ME installed along with its dependencies. Instructions for setting up the Docker container with CGRA-ME installed are in the lab install instruction handout.

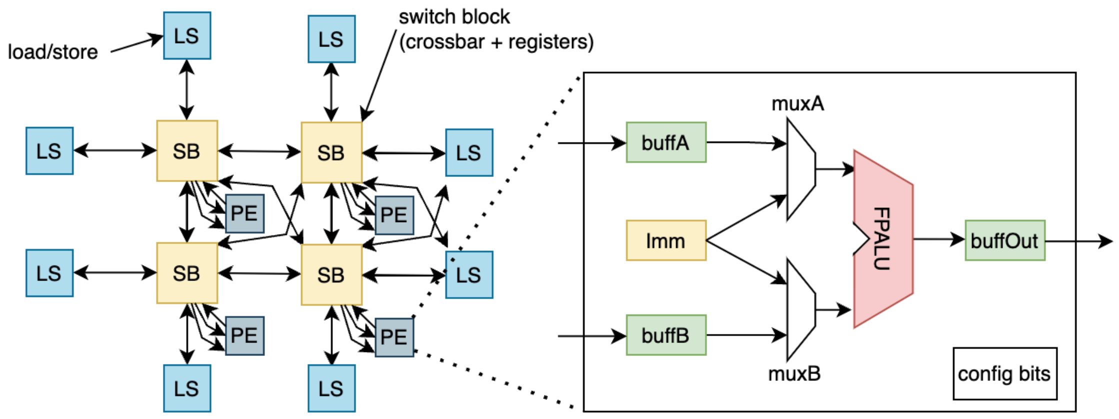

Figure 2: The Riken CGRA architecture.

If the lab set-up instructions were followed, and the docker container state saved, it can be restored by simply running the docker container again with the following command:

```
sudo docker run -it -v /sys:/sys:ro -v \$(pwd):/build cgrame bash
```

If the docker container state was not saved or CGRA-ME was not built, you will have to build it now as done in the lab setup instructions. Once set up, navigate to /tmp/cgra_me/cgra-me-release-2.0.2 and set up the environment with the following command:

```
./cgrame_env
```

You can now view the list of CGRA-ME commands with:

```
cgrame -h
```

The lab will involve modifying and exploring files inside the container. If you're comfortable with `vim` you may use that, but if you'd prefer an easier development environment, consider installing Dev Containers and using Remote Explorer with VSCode to explore the container.

1 Mapping One Application to an Architecture

To begin with, let's map one application to the RIKEN CGRA. The architecture is shown in Fig. 2. The application performs a stencil computation (like a 2D convolution). In CGRA-ME, applications are described using DOT format. Navigate to the `date_tut` directory to view the benchmarks to be used in this lab. Use the following command to navigate to the directory:

```
cd /tmp/cgra_me/cgra-me-release-2.0.2/date24_tut
```

First, let's look at the application visually. Look at the contents of the file `Stencil_unroll2.dot` and copy this to your clipboard. In a web browser, visit:

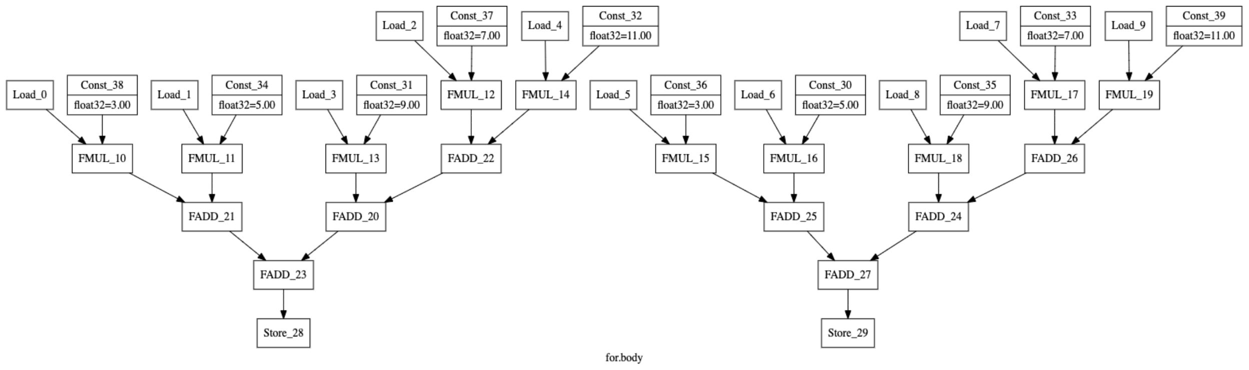

Figure 3: Dataflow graph for the stencil application.

`https://dreampuf.github.io/GraphvizOnline/`. Paste the clipboard contents into the left panel of the browser window. You will see the dataflow graph shown in Fig. 3, containing floating point additions, multiplications and constants, as well as I/O.

Before mapping, find out the list of supported architectures in CGRA-ME 2.0, type:

```
cgrame --arch-list
```

You will see the following output:

```
List of C++ CGRA Architectures: [ID: description]
1: Adres CGRA Architecture
2: HyCUBE CGRA Architecture
3: Elastic RIKEN Architecture
```

Note that architecture #3 is the RIKEN architecture.

The mapper requires three inputs: a dataflow graph for the application, scheduling constraints, and a device model of the architecture. Note that the RIKEN CGRA is an elastic CGRA, hence it does not require scheduling constraints. The mapping flow can be seen in Fig. 4

Now, map the application to the RIKEN CGRA with 5 rows and 5 columns using the following command:

```
cgrame -g Stencil_unroll2.dot -c 3 -m ClusteredMapper --arch-opts
"rows=5 cols=5" > log.out
```

Let's briefly dissect the meaning of the various parts of the above command. `cgrame` is the the CGRA-ME binary. `-g Stencil_unroll2.dot` indicates the application file name. `-c 3` means to target the RIKEN CGRA. `-m ClusteredMapper` indicates which mapping algorithm should be used, in this case, the clustered mapper [3]. `-arch-opts "rows=5 cols=5"` indicates the number of CGRA rows and columns. Lastly, `> log.out` redirections the output of the mapping to a log file.

Open the `log.out` file using your favourite editor. Find the line that contains:

```
[INFO] Mapping Success: 1
```

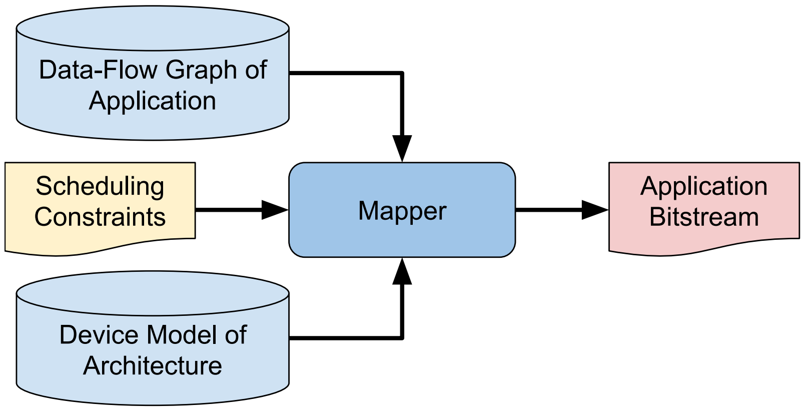

Figure 4: An overview of the CGRA-ME mapping flow.

It should be around line 978 of the `log.out` file. Below this line, you will see lines like this:

```
Operation Mapping Result:
Const_30(const):   0:pe_c3_r4.Imm.const
Const_31(const):   0:pe_c2_r4.Imm.const

...

FADD_20(fadd):   0:pe_c1_r4.FuncUnit.FuncUnit_FPALU.fpunit
FADD_21(fadd):   0:pe_c1_r2.FuncUnit.FuncUnit_FPALU.fpunit
```

These lines give the placement of the operations in the CGRA. For example, `Const_30` from the stencil application is placed in the processing element (PE) at column 3, row 4. The `FADD_20` is placed in the PE at col 1, row 4.

Below the placement results, you will find a line like this that shows the routing resources that are used for each signal in the application:

```
    Connection Mapping Result:
```

After this line, you will find a set of lines that look like this:

```
Const_39_out:
  0:pe_c2_r0.muxA.mux
  0:pe_c2_r0.muxA.out
  0:pe_c2_r0.FuncUnit.FuncUnit_join.out0
  0:pe_c2_r0.FuncUnit.FuncUnit_FPALU.in_a
  0:pe_c2_r0.FuncUnit.FuncUnit_join.in0
  0:pe_c2_r0.FuncUnit.in_a
  0:pe_c2_r0.Imm.out
  0:pe_c2_r0.Imm_fork.in
  0:pe_c2_r0.Imm_fork.fork
  0:pe_c2_r0.Imm_fork.out0
  0:pe_c2_r0.muxA.in1

FADD_20_out:
  0:pe_c1_r3.crossbar.in2
  0:pe_c1_r3.crossbar.fork_crossbar2.in
  0:pe_c1_r3.crossbar.fork_crossbar2.fork
  0:pe_c1_r3.crossbar.fork_crossbar2.out9
  0:pe_c1_r3.crossbar.mux_9.in2
  0:pe_c1_r3.reg2.reg
  0:pe_c1_r3.reg2.out
  0:pe_c1_r3.reg2.m_out
  0:pe_c1_r3.crossbar.mux_9.mux
  0:pe_c1_r3.muxB.in0
  ...
```

Observe that for `Const_39_out`, all the routing is contained within the PE at column 2, row 0. The constant value is routed through `muxA` in the PE to reach input A of the floating-point ALU.

By exploring these sections of the `log.out` file, we can understand and analyze the placement and routing solution produced by the mapper for the application.

Here, we set the number of rows and columns of the CGRA on the command line. If you're interested in the other architectural parameters that can be changed for the RIKEN architecture, type:

```
cgrame --arch-opts-list 3
```

2 Mapping a Set of Applications to an Architecture

In this exercise, you will map three applications to the RIKEN CGRA, considering four CGRA different array sizes, ranging from 3×3 to 6×6. Navigate to the directory: /tmp/cgra_me/cgra-me-release-2.0.2/date24_tut

List out the contents of the directory and you will find three applications: convolution, stencil, and FFT. Now, in the same directory, map the three applications to the RIKEN CGRA, while varying the CGRA dimensions. We have created a PERL script to automate this:

```
perl run.pl
```

If you look at the script contents, you'll see it contains loops, and the same command you used above to map an application onto the RIKEN CGRA. When the script completes, it reports, for each of the 3 applications, and for each CGRA size, whether the mapping was successful. Feel free to check the contents of `run.pl` to see what it does by reading the comments. In the table belo0w (or your own notebook), record the mapping success for each of the three benchmarks. You *should* observe that some of the benchmarks cannot be mapped to the smaller CGRA array sizes; rather, they can only be mapped to larger CGRAs.

Benchmark	**CGRA size**	**Mapping Success?**
conv_unroll2.dot	3x3	
conv_unroll2.dot	4x4	
conv_unroll2.dot	5x5	
conv_unroll2.dot	6x6	
Stencil_unroll2.dot	3x3	
Stencil_unroll2.dot	4x4	
Stencil_unroll2.dot	5x5	
Stencil_unroll2.dot	6x6	
fft_radix4.dot	3x3	
fft_radix4.dot	4x4	
fft_radix4.dot	5x5	
fft_radix4.dot	6x6	

3 Modifiying an Architecture

Now we will modify the RIKEN architecture to split the ALU into two ALUs: one capable of add/subtract, and the second capable of multiply. The new tile is shown in Fig. 5. Observe that to make this architecture change, we will need to add a second ALU. As well, we will need to add new inputs/outputs to the switch block (SB). In essence, this increases the logic density of the RIKEN architecture by allowing two ALU operations to be placed in each tile (vs. just one ALU operation in the original RIKEN CGRA). This architecture change has been proposed in this paper [1].

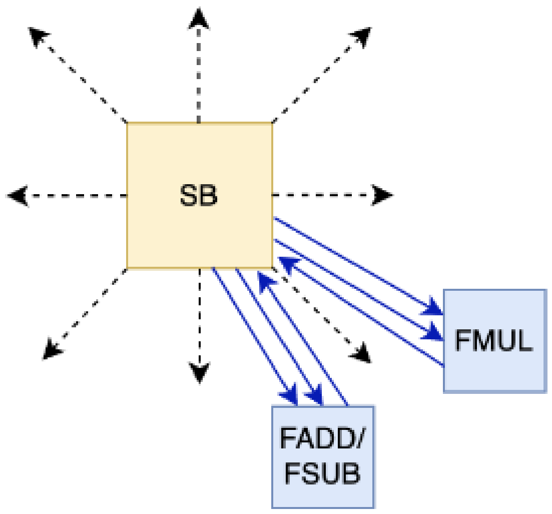

Figure 5: Modified Riken CGRA architecture with two ALUs per tile.

Navigate to this source code directory: `../cgra-me/src/modules`

Open the RIKEN PE architecture model in this C++ file: `RIKENPE_Elastic.cpp`

There are **four** code changes to implement the new architecture. Please follow carefully!

3.1 Code Change #1 – Change the Function of First ALU

Around line 55 of the file you'll find the code instantiating the components within the Riken PE:

```
  // **************** INSTANTIATE SUBMODULES OF THE PE ************
  // Add the "guts" of the PE, e.g., the elastic buffers, constant reg, ALU, etc.

  addSubModule(new ElasticBufferFifo("buffA", loc, buffer_depth, 32, eb_enable), ...);
  addSubModule(new ElasticBufferFifo("buffB", loc, buffer_depth, 32, eb_enable), ...);
  addSubModule(new ConstUnit("Imm", loc, 32, 1, isElastic), ...);
  ...

  addSubModule(new ElasticFPUnit("FuncUnit", loc, {
        OpCode::FADD,
        OpCode::FMUL,
        OpCode::FDIV,
        OpCode::SQRT,}, size, II, latency, pred, isElastic), 0.5, 0.4, 0.5, 0.1);
```

Change the ALU so it cannot perform floating-point multiply. To do this comment out the line referring to FMUL using the `//`. The modified code should look like this:

```
    //  OpCode::FMUL,
```

3.2 Code Change #2 – Adding Another ALU

Continuing down the file, around line 135 you will see code like this that will instantiate a second ALU. Notice that the second ALU cannot perform addition; it can only perform multiplication.

```
#if 0 // SECOND ALU
    // *************** INSTANTIATE SUBMODULES OF THE PE ************
    // Add the "guts" of the PE, e.g., the elastic buffers, constant reg, ALU, etc.

    addSubModule(new ElasticBufferFifo("buffA_2", loc, buffer_depth,
    32, eb_enable), ...);
    addSubModule(new ElasticBufferFifo("buffB_2", loc, buffer_depth,
    32, eb_enable), ...);
    addSubModule(new ConstUnit("Imm_2", loc, 32, 1, isElastic), ...);
```

Change the line that begins `#if 0 // SECOND ALU`. It should now appear like this:

```
      #if 1 // SECOND ALU
```

This change will instantiate the second ALU, including the buffers on its inputs and output, the internal fork, and so on.

3.3 Code Change #3 – Add Extra SB Inputs/Outputs

Further down in the code, look for these lines:

```
#if 0 // SECOND ALU
    numCrossbarInputs += 1; // extra input for second ALU
    numCrossbarOutputs += num_ALU_inputs;  // extra outputs for second ALU
#endif
```

Change the compiler directive line: `#if 0` to be: `#if 1`. This will increase the number of SB inputs and outputs to accommodate the new ALU.

3.4 Code Change #4 – Connect the SB to the Second (New) ALU

Finally, right at the end of the file, you'll see this:

```
#if 0 // SECOND ALU
    // buffOut to crossbar
    connectPorts("buffOut_2.data_out", "crossbar.in" +
    std::to_string(num_inPorts + 1), isElastic);
    // Crossbar to buffA and buffB
    connectPorts("crossbar.out" + std::to_string(num_outPorts + 2),
    "buffA_2.data_in", isElastic);
    connectPorts("crossbar.out" + std::to_string(num_outPorts + 3),
    "buffB_2.data_in", isElastic);
#endif
```

Again, change the compiler directive from `#if 0` to be `#if 1`. This code makes the connections between the new SB inputs/outputs to the new ALU's output/inputs.

3.5 Recompile CGRA-ME

Navigate to the main CGRA-ME directory: `./cgra-me/` and type:

```
    make
```

CGRA-ME will be recompiled.

Congratulations you have made your first architecture change in CGRA-ME!

3.6 Re-Mapping the Applications

Now that you have two ALUs per switch block, re-map the applications by using `run.pl` as outlined in Section 2 and fill the table below.

What changes do you observe relative to the original RIKEN architecture? You should see that, with the second ALU present, some of the applications can be mapped into CGRAs with *smaller* array sizes.

Benchmark	**CGRA size**	**Mapping Success?**
conv_unroll2.dot	3x3	
conv_unroll2.dot	4x4	
conv_unroll2.dot	5x5	
conv_unroll2.dot	6x6	
Stencil_unroll2.dot	3x3	
Stencil_unroll2.dot	4x4	
Stencil_unroll2.dot	5x5	
Stencil_unroll2.dot	6x6	
fft_radix4.dot	3x3	
fft_radix4.dot	4x4	
fft_radix4.dot	5x5	
fft_radix4.dot	6x6	

4 C-Code Application Flow

Another useful functionality of CGRA-ME is its ability to generate data flow graphs (DFGs) directly from C-code applications. By providing a C-code application as input, CGRA-ME converts it into a data flow graph, which integrates into the CGRA-ME workflow.

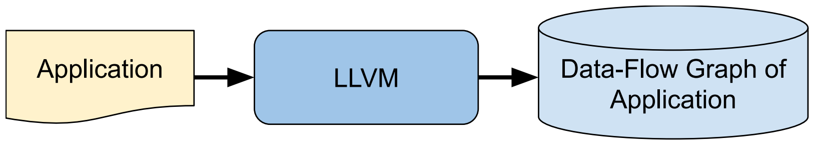

Figure 6: An overview of the LLVM-based DFG Generation Flow.

4.1 Generating a DFG

In this section, we'll walk through how to generate a DFG from a C application. Various benchmarks can be found in the `cgra_me/benchmarks/microbench` directory. There are pre-generated data flow graphs along with the associated C applications. To generate a new DFG, we'll do the following:

```
cd ./cgra_me-release-2.0.2/benchmarks/microbench/sum

make
```

This will generate a DOT file with the application's data flow graph. Before this DFG is mapped to a CGRA architecture, CGRA-ME will first perform some legalization and optimization steps. We can view the final DFG with the following command:

```
cgrame -g graph_loop.dot --exit-after-optimizing-dfg
--print-final-dfg final_graph_loop.dot
```

Now that we have the final DFG try exploring the original `sum.c` file and the generated `final_graph_loop.dot`. Remember, you can use Graphviz to visualize the DOT file. You should see that there is a Phi operation, which handles the loop index, a load to grab our value, and the value is summed and output.

4.2 Simple DFG Modifications

Now, if we wish to write a new DFG graph or modify a current benchmark, we do not have to walk through and modify the generated `graph_loop.dot`, we can simply apply the required changes to the C application.

Try adding another input to sum, so that we sum a and b. This can be done by adding the following line at the start:

```
static int* b = (int*)(2*sizeof(int));
```

And modifying the loop:

```
for (i = 0; i < N; i++) {
    //DFGLoop: loop
    sum += a[i] + b[i];
}
```

Remove graph_loop.dot by running

```
rm graph_loop.dot
```

Then try generating the DFG again. Explore the results and see how our changes modified the DFG.

5 Functional Verification

For the functional simulation, we utilize a different CGRA architecture called HyCUBE [2][1].

5.1 Generating RTL of an Architecture

Another aspect of CGRA-ME we want to explore is generating an architecture's Register Transfer Level (RTL) description. The architecture generation flow can be seen in Figure 7. To generate RTL code for your own CGRA, you need to create an architecture description using the CGRA-ME C++ API.

As seen above, you can view all the CGRA-ME architectures currently available in CGRA-ME with the following command:

```
cgrame --arch-list
```

[1]The reason for targeting the HyCUBE CGRA is that the floating-point cores in the RIKEN CGRA are specified in VHDL and the CGRA itself is in Verilog, requiring a mixed-language simulator.

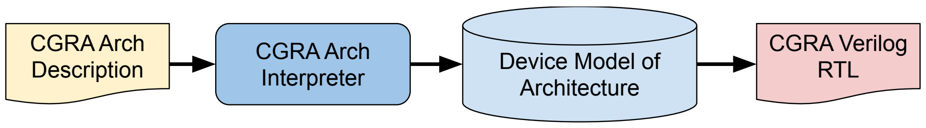

Figure 7: An overview of the CGRA-ME architecture generation flow.

Navigate back to: `/tmp/cgra_me/cgra-me-release-2.0.2/date24_tut/sim`. To generate the RTL for one of the architectures listed. We must specify certain architectural parameters. The following command will generate a 2×2 HyCUBE CGRA:

```
cgrame -c hycube --arch-opts "rows=2 cols=2"  --gen-verilog .
```

In the generated file `cgrame.v`, locate `cgra_U0` at line `:2063`. This module represents the top-level entity, instantiating the PE-Matrix, IO, and memory ports of the CGRA, along with interconnections. Subsequently, at line `:2328`, you'll find the first instance of the PE module `pe_c0_r0`, named `hycube_in15_out15_U3`. Its definition is available at line `:1411`, adhering to the C++ architecture defined by the user.

5.2 ModelSim Simulation

Now that we have learned how to generate the RTL of the architecture in Section 5.1 and generate a bitstream in Section 1. We wish to combine the two files to carry out a functional simulation. Figure 8 shows the simulation flow.

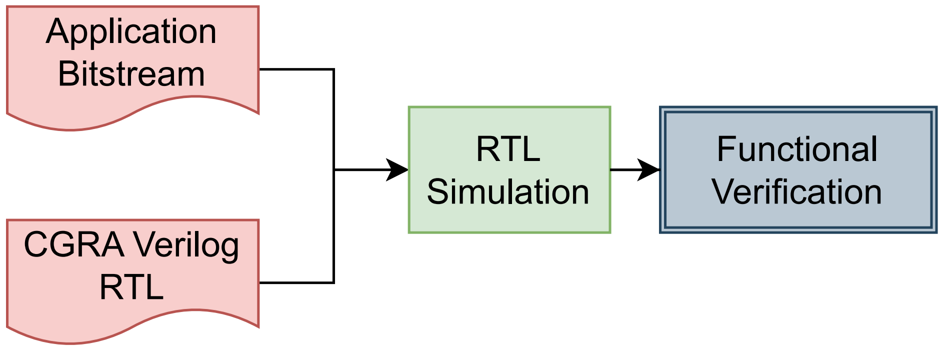

Figure 8: An overview of the functional simulation flow.

Next, let's walk through how to carry out a functional simulation. While in the `sim` folder in `date24_tut` directory in which we generated the CGRA RTL. In this directory, we have the DFG of the `sum` benchmark and `tb_master.sv`, which is the testbench file. Now, we want to generate both the bitstream file in this directory using the following command:

```
cgrame -g graph_loop.dot -c hycube -m ClusteredMapper --arch-opts
"rows=2 cols=2" --gen-testbench . > log.out
```

Now that we have the three required files to run the simulation which are:

- `cgrame.v`: CGRA RTL

- `testbench.v`: application bitstream
- `tb_master.sv`: simulation file that configures the generated CGRA RTL using the generated bitstream

We can run a functional simulation by running the following commands:

```
/opt/altera/modelsim_ase/bin/vlib work
/opt/altera/modelsim_ase/bin/vlog testbench.v
/opt/altera/modelsim_ase/bin/vlog cgrame.v
/opt/altera/modelsim_ase/bin/vlog tb_master.sv
/opt/altera/modelsim_ase/bin/vsim -c tb_master -do "run -all"
```

The expected output is the following:

```
# ** Note: $finish    : tb_master.sv(206)
#    Time: 345300 ps  Iteration: 0  Instance: /tb_master
# End time: 06:14:52 on Mar 18,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
```

If you explore the `tb_master.sv` initial block you are going to notice that it starts with configuring the CGRA in this line:

```
while (1) if (configurator_done) break; else #1;
```

After which it checks the output of the benchmark on given cycles with golden output at line `:194`. If the output is incorrect it will output `test failure`; otherwise, it gracefully exits.

References

[1] Jason Anderson, Boma Adhi, Carlos Cortes, Emanuele Del Sozzo, Omar Ragheb, and Kentaro Sano. Exploration of compute vs. interconnect tradeoffs in CGRAs for HPC. In *HEART*, page 59–68, 2023.

[2] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *IEEE/ACM DAC*, 2017.

[3] Omar Ragheb and Jason Heldge Anderson. CLUMAP: Clustered mapper for CGRAs with predication. In *ACM/IEEE DAC*, 2024.

[4] Omar Ragheb, Stephen Wicklund, Matthew Walker, Rami Beidas, Adham Ragab, Tianyi Yu, and Jason Anderson. CGRA-ME 2.0: A research framework for next-generation CGRA architectures and CAD. In *Int'l Workshop on CGRAs for High-Performance Computing (CGRA4HPC)*, 2024.