GNU Radio Out-of-Tree Modules with OPINCAA

Flueratoru Laura

1 Introduction

This report describes the process of creating a GNU Radio out-of-tree module which interfaces with OPINCAA[1], the framework used to program the ConnexArray SIMD[2]. At the moment, the module works only with the simulator, but support for the hardware will be added in the future. The report is organised as follows: §2 describes the general structure of an out-of-tree module and the main files that it contains, while in §3 lie the steps needed to actually create the module which performs a simple addition (without using OPINCAA) and testing it using Python. Section §4 presents the changes needed in order to create an OPINCAA kernel with a similar functionality inside GNU Radio, how to write a C++ test for it and running it using the ConnexArray simulator. Finally, in §5 I present some conclusions and future work plans and in §6 can be found the code snippets to support the explanations throughout the report.

2 Structure of an out-of-tree module

A module should contain the following directories and files:

- apps/
- build/
- cmake/
- CMakeLists.txt
- docs/
- examples/
- grc/
- include/
- lib/

- python/
- swig/

Although **gr_modtool** takes care of this structure and creates most of the files for us, it is worthwhile mentioning some of the most important files, where they are found and some coding details about them[3], [4].

2.1 Coding structure

Public Header Files

- Defined in **include/foo**
- Installed into **prefix/include/foo**

Private Implementation Header Files

- Defined in **lib/**
- Are relevant only for the compilation, but not installed later
- This is according to the pImpl idiom, where we hide the implementation details of an interface, in order to avoid recompilation of the modules using it in case it is changed[5].

Implementation Source Files

- Located in **lib/**

SWIG Interface File

- A single file located in the directory **swig/**, where we include the headers in the main interface file
- SWIG (Simplified Wrapper and Interface Generator) is used, in this case, to connect code written in C++ with one written in Python, so modules written in C++ are still available in Python[6].

Python Files

- Located in the **python/** directory
- Contains unit tests (not installed) and parts of the Python module (installed)

XML Scripts

- Located in the **grc/** directory
- Needed in order to make the out-of-tree module created available in the GUI called GNU Radio Companion[7].

CMakeLists

- There is one **CMakeLists.txt** in every subdirectory of the OOT module.
- Contain instructions about how to find the libraries needed by our module.

3 Steps taken to create a new out-of-tree module

- Used the **gr_modtool** script to create a new module named **gr-opincaa**. This creates the directory structure and some of the files specified above, using the following command:

```
gr_modtool newmod opincaa
```

- Created a block that adds two integers on the Connex SIMD, named **add_ii**. For this step, we also use **gr_modtool**, specifying it to create a new module which has a 1:1 relationship between the input rate and the output rate, specified by the parameter `sync`:

```
gr_modtool add -t sync add_ii
```

  Additionally, I added Python QA support to be able to create tests for the block. Also, I foresaw a step needed for creating a ConnexMachine and when asked to "Enter valid argument list, including default arguments:", I specified the following parameters:
  `std::string distributionFIFO`, `std::string reductionFIFO`, `std::string writeFIFO` and `std::string readFIFO`.

  This is because the ConnexMachine object needs to receive as parameters the FIFO queues that help it to communicate with the SIMD processor or the simulator. Since we pass the paths as parameters, we do not need to take care of both of the cases in the worker and, should the paths ever change, we need only to call the tester with the right arguments.

- The **add_ii.cc** file needs to be modified to fit with its purpose. Since we want to implement an adder, we need two input ports and one output port, so the constructor of the block needs to have the following structure:

```
add_ii_impl::add_ii_impl(
    std::string distributionFIFO,
    std::string reductionFIFO,
    std::string writeFIFO,
    std::string readFIFO
    )
  : gr::sync_block
  ("add_ii",
    gr::io_signature::make(2, 2, sizeof(int)),
    gr::io_signature::make(1, 1, sizeof(int)))
{}
```

  The `io_signature` method takes information about the minimum number of ports, the maximum number of ports and their size, respectively. Also, we notice that **gr_modtool** added the specified parameters as arguments to the constructor.

- Fot the beginning, in the program I simply added two numbers using C++ and wrote a test for it, to make sure that it is working accordingly. The processing is done in the worker, where `input_items[i]` and `output_items[i]` are arrays which contains the data of the `i`-th input and output, respectively.

  When using the `work` method, the length of the input arrays is equal with the one of the output, so we iterate over `noutput_items` to compute the sum. The full code of the worker function can be found in the subsection §6.1.

- Since we want to ensure that the program is running correctly, we must write a test for it. The code can be found in the subsection §6.2

  We define two input arrays and another one that contains the output we would expect. Then, we source the elements of both of the inputs, state that `opincaa.add_ii` is the block that we are testing and gather its output with `blocks.vector_sink_i`. Afterwards we build the graph, run it and compare the results.

```
Test project /home/laoo/work/licenta/gr-opincaa/build
    Start 3: qa_add_ii
1/1 Test #3: qa_add_ii ..............   Passed    0.16 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.16 sec
```

Since the test passes, we can proceed to add the OPINCAA kernel.

4 Add support for OPINCAA in GNU Radio

The following changes need to be made to the block:

- I chose to keep a pointer to the `ConnexMachine` object as a private member of the `add_ii` block. Then, since we need only an instance of it, we create it in the constructor of the class §6.3, using the arguments that contain the paths to the FIFO queues.
- In the worker, we define a simple `add` kernel that initializes R1 and R2 with the contents of, respectively, the first and second vectors of the local storage, then adds their contents and stores the result in R2. Finally, a `reduce` operation is applied to all the R2 registers.
- Before executing the kernel, we write the two inputs to the first two vectors of the local storage using the method `writeDataToArray`. Finally, we run the kernel and assign the result of the reduction to the first element of the output vector.

The full source code of this function can be found in section §6.4.

There were several problems at compile time. Firstly, the `ConnexMachine.h` header needs to be added to the **add_ii_impl.h** file. Because this header uses certain features from C++11, support for this standard must be added, so the line

```
1 SET(CMAKE_CXX_FLAGS "\${CMAKE_CXX_FLAGS} -std=c++0x")
```

needs to be written in the **CMakeLists.txt** file, in the **Compiler Specific Setup** section.

Then, the `opincaa` library needs to be added to the `GR_SWIG_LIBRARIES` in **swig/CMakeLists.txt**, in order to be able to link the symbols related to it.

Even with these changes, I was unable to use the Python unit testing, as even though the ConnexMachine was created and the kernel was succesfully executed, giving correct results, right before returning the `noutput_items` in the `work` method, the thread caught an unrecognized exception. I suspect the problem comes from the way in which SWIG connects the C++ modules with the Python tests and this is an issue I will further investigate, as I would like to be able to use the unit testing framework with Python.

Given this issue, I chose to create a C++ test to verify if I would encounter the same problem. I used a similar stucture of the testing code as I did in the Python test, as can be seen in the subsection §6.5.

The next step was to create a Makefile to build the block, as well as the test (§6.6). Also, in order to make the testing easier, I created a bash script to run the simulator and the test with the correct parameters. It is worthwhile mentioning that, currently, the OPINCAA project must be installed separately (and the simulator for the ConnexArray SIMD must be also built along) and the relative (or absolute) path to the project must be set before running the test. A better approach would be adding the OPINCAA project directly as a git submodule for the current module, in order to ensure that all the dependencies are satisfied and to eliminate the need to modify the path.

By running `make` and then `./test_simulator.sh` I get the following output, which means that the program is functioning correctly:

```
laoo@neverland ~/work/licenta/gr-opincaa $ ./test_simulator.sh
FIFO distributionFIFO succesfully opened!
FIFO writeFIFO succesfully opened!
FIFO reductionFIFO succesfully opened!
FIFO readFIFO succesfully opened!
Starting Core Thread...
Starting IO Thread...
/home/laoo/work/licenta/gr-opincaa/distributionFIFO
/home/laoo/work/licenta/gr-opincaa/reductionFIFO
/home/laoo/work/licenta/gr-opincaa/writeFIFO
/home/laoo/work/licenta/gr-opincaa/readFIFO
ConnexMachine created !
Reduction = 105
./test_simulator.sh: line 38: 13175 Killed
```

5 Conclusion

The report presented the way in which a custom out-of-tree GNU Radio module can be created in order to be used with a ConnexArray machine and, subsequently, with the OPINCAA framework. The present code can be used as a base to create complex kernels that implement signal processing blocks on the SIMD accelerator.

As future improvements, I plan on solving the issue with the QA Python environment provided by GNU Radio, as it is desirable to have a unit testing framework rather than building the tests separately and Python is a better programming language choice for this task than C++. Another thing to do

would be to add the OPINCAA project as a git submodule of the current one, at least until a better option is available, since the OOT module depends on it for the simulator.

6 Code snippets

6.1 Work Method: add_ii_impl.cc

```
int
add_ii_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  const int *in0 =
    reinterpret_cast<const int *>(input_items[0]);
  const int *in1 =
    reinterpret_cast<const int *>(input_items[1]);
  int *out = reinterpret_cast<int *>(output_items[0]);

  for(size_t i = 0; i < noutput_items; i++) {
    out[i] = in0[i] + in1[i];
  }

  return noutput_items;
}
```

6.2 Python test: qa_add_ii.py

```
def test_001_t (self):
  src1_data = (-1, 0, 2, 4, 100)
  src2_data = (1, 20, -5, 100, 4)
  exp_result = (0, 20, -3, 104, 104)

  src1 = blocks.vector_source_i(src1_data)
  src2 = blocks.vector_source_i(src2_data)

  adder = opincaa.add_ii()

  dst = blocks.vector_sink_i()

  # connect the adder with its multiple inputs
  self.tb.connect(src1, (adder, 0))
  self.tb.connect(src2, (adder, 1))

  # connect the adder to the sink
  self.tb.connect(adder, dst)

  # run the graph
  self.tb.run ()

  # check data
  result_data = dst.data()
  self.assertTupleEqual(result_data, exp_result)
```

6.3 Constructor of the class: add_ii_impl.cc

```
add_ii_impl::add_ii_impl(
    std::string distributionFIFO,
    std::string reductionFIFO,
    std::string writeFIFO,
    std::string readFIFO
    )
  : gr::sync_block("add_ii",
          gr::io_signature::make(2, 2, sizeof(int)),
          gr::io_signature::make(1, 1, sizeof(int)))
{
  try {
    connex = new ConnexMachine(distributionFIFO,
                               reductionFIFO,
                               writeFIFO,
                               readFIFO);
  } catch (std::string err) {
    std::cout << err << std::endl;
  }
}
```

6.4 Work method using an OPINCAA kernel: add_ii_impl.cc

```
int
add_ii_impl::work(int noutput_items,
    gr_vector_const_void_star &input_items,
    gr_vector_void_star &output_items)
{
  if (!connex) {
    // the ConnexMachine was not successfully created
    return noutput_items;
  }

  const int *in0 = reinterpret_cast<const int *>(input_items[0]);
  const int *in1 = reinterpret_cast<const int *>(input_items[1]);
  int *out = reinterpret_cast<int *>(output_items[0]);

  BEGIN_KERNEL("add");
    EXECUTE_IN_ALL(
                    R1 = LS[0];
                    R2 = LS[1];
                    R2 = R1 + R2;
                    REDUCE(R2);
                  )
  END_KERNEL("add");

  connex->writeDataToArray(in0, noutput_items, 0);
  connex->writeDataToArray(in1, noutput_items, 1);

  connex->executeKernel("add");
  out[0] = connex->readReduction();

  return noutput_items;
}
```

6.5 C++ test

```
#include <gnuradio/top_block.h>
#include <gnuradio/blocks/vector_source_i.h>
#include <gnuradio/blocks/vector_sink_i.h>
#include "include/opincaa/add_ii.h"

using namespace gr;

int main(int argc, char **argv)
{
  top_block_sptr tb = make_top_block("add_connex");

  // Define the input data
  const std::vector<int> src1_data = {1, 2, 3, 4, 5, 6, 7};
  const std::vector<int> src2_data = {8, 9, 10, 11, 12, 13, 14};

  // Create two source blocks that pass the input data
  blocks::vector_source_i::sptr src1 =
      blocks::vector_source_i::make(src1_data);
  blocks::vector_source_i::sptr src2 =
      blocks::vector_source_i::make(src2_data);

  // Create an add_ii block - takes as arguments the FIFO paths
  opincaa::add_ii::sptr adder = opincaa::add_ii::make(
      argv[1], argv[2], argv[3], argv[4]
      );

  // Create a sink block
  blocks::vector_sink_i::sptr dst = blocks::vector_sink_i::make();

  // Connect the blocks
  // Connect the output of src1 to the adder's first input
  tb->connect(src1, 0, adder, 0);

  // Connect the output of src2 to the adder's second input
  tb->connect(src2, 0, adder, 1);

  // Connect the adder's output to the sink's input
  tb->connect(adder, 0, dst, 0);

  // Run the flow graph
  tb->run();

  // Check the data obtained
  std::vector<int> dst_data = dst->data();
  std::cout << "Reduction = " << dst_data[0] << std::endl;

  return 0;
}
```

6.6 Makefile

```
WD = $(shell pwd)
CXXFLAGS = -g -std=c++11
LDFLAGS = -L/usr/lib -L/usr/local/lib/ -Lbuild/lib/
LDLIBS = -lgnuradio-runtime -lgnuradio-opincaa -lopincaa \
  -lboost_system  -lgnuradio-blocks -lgnuradio-pmt

all: connex_adder

build/Makefile: CMakeLists.txt
  mkdir -p build && cd build && cmake ..

build/lib/gnuradio-opincaa.so: build/Makefile
  cd build && make

connex_adder: test_add_block.cc build/lib/gnuradio-opincaa.so
  g++ $< $(LDFLAGS) $(CXXFLAGS) $(LDLIBS) -o $@

clean:
  rm connex_adder

distclean:
  git clean -xfd
```

6.7 Testing script

```
#!/bin/bash

# Quick workaround: be sure to set the correct path to OPINCAA
OPINCAA_PATH=../../opincaa

SIM_PATH=$OPINCAA_PATH/simulator
GR_OPINCAA_LIB=build/lib/
CONNEX16_LIB=$OPINCAA_PATH/libs/connex16-hm-generic

FIFO_ROOT_PATH=/home/laoo/work/licenta/gr-opincaa
DISTRIBUTION_FIFO=$FIFO_ROOT_PATH/distributionFIFO
REDUCTION_FIFO=$FIFO_ROOT_PATH/reductionFIFO
WRITE_FIFO=$FIFO_ROOT_PATH/writeFIFO
READ_FIFO=$FIFO_ROOT_PATH/readFIFO

#start the simulator
./$SIM_PATH/build/simulator &

#run the test
sleep 3
LD_LIBRARY_PATH=$CONNEX16_LIB:$GR_OPINCAA_LIB
export LD_LIBRARY_PATH

# run C++ test
./connex_adder $DISTRIBUTION_FIFO $REDUCTION_FIFO $WRITE_FIFO
$READ_FIFO

#kill simulator
killall -9 simulator
```

References

[1] http://www.imt.ro/romjist/Volum16/Number16_4/pdf/06-LPetrica.pdf

[2] is there any paper about it?

[3] http://gnuradio.org/redmine/projects/gnuradio/wiki/BlocksCodingGuide

[4] http://gnuradio.org/redmine/projects/gnuradio/wiki/OutOfTreeModules

[5] https://en.wikipedia.org/wiki/Opaque_pointer

[6] http://www.swig.org/exec.html

[7] http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion