# GNU Radio 4.0

## Overview and Block Migration

# Outline

- Creating a Block
  - Development Methodology
  - Block API
  - Multiple Implementations
  - Python Blocks
  - History
  - Forecast
- Custom Buffers
  - Overview
  - Differences from GR 3.10
- Future Plans

# Feedback Welcome!

At this stage in the development cycle, we are happy to entertain even large changes to things

What is in dev-4.0 is currently a good starting point for GR 4.0.0, but now is the time to make aggressive changes

Please submit PRs!!!!!

File issues starting with 4.0: ...

# Setting Up 4.0 Environment

1) Install prerequisites
2) Create prefix
3) Clone gnuradio --branch dev-4.0
4) Build/install

Tutorial Code can be found:

https://github.com/mormj/gr4-grcon22

# Exercise 1: Creating OOT with a block

Doing this: https://wiki.gnuradio.org/index.php?title=Creating_c%2B%2B_OOT_with_gr-modtool

... but with 4.0

1) Create OOT

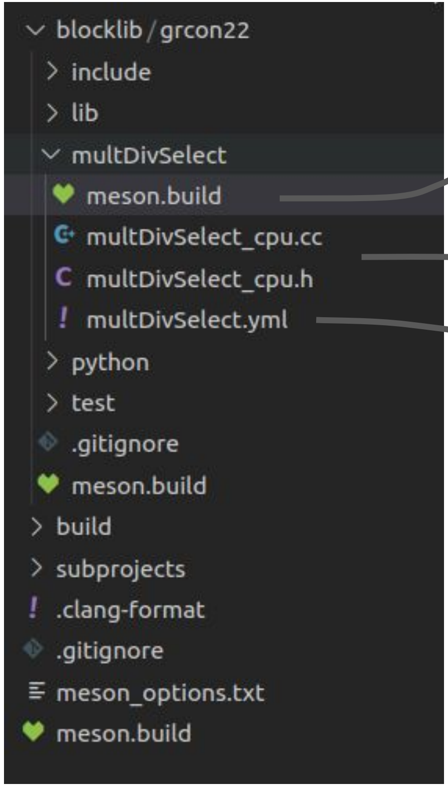
```
cd $GR_PREFIX && source setup_env.sh
cd $GR_PREFIX/src
python3 $GR_PREFIX/src/gnuradio/utils/modtool/create_mod.py grcon22
```

2) Create Block

```
cd gr4-grcon22
python3 $GR_PREFIX/src/gnuradio/utils/modtool/create_block.py --templated multDivSelect
```

Now, let's look at folder and file structure …

# Block folder structure




(autogenerated) replaces CMakeLists.txt

Block implementation for cpu flavor of block

Our starting Point

# The Block .yml

```yaml
module: grcon22
block: multDivSelect
label: multDivSelect
blocktype: sync_block
```

```yaml
# Example Parameters
parameters:
-   id: k
    label: Constant
    dtype: typekeys/T
    settable: true
-   id: vlen
    label: Vec. Length
    dtype: size
    settable: false
    default: 1
```

```yaml
typekeys:
  - id: T
    type: class
    options:
          - cf32
          - rf32
          - ri32
          - ri16
          - ri8
```

```yaml
implementations:
-   id: cpu
# -   id: cuda
```

```yaml
ports:
-   domain: stream
    id: in
    direction: input
    type: typekeys/T
    shape: parameters/vlen

-   domain: stream
    id: out
    direction: output
    type: typekeys/T
    shape: parameters/vlen
```

**Modtool can remain simple because this file is editable**

# The block properties

```
module: grcon22  # should not change

block: multDivSelect  # should not change

label: Mult/Div Select  # how does it show up in GRC

blocktype: sync_block  # can also be "block"
```

# Typekeys

- Use sigmf-like nomenclature for the types (https://github.com/gnuradio/SigMF/blob/sigmf-v1.x/sigmf-spec.md#sigmf-dataset-format)
- Templating allows a block to have multiple possible instantiations with different port types - with a lot less effort than that took in GR 3.x

```
typekeys:
  - id: T             # Can be anything, but is referenced from port section
    type: class       # how it gets instantiated in C++
    options:          # For this block, let's just do float and complex
        - cf32
        - rf32
```

# Parameters

Parameters become some combination of constructor arguments and a PMT object accessible thread-safe from the work function

```
parameters:
-   id: select
    label: Select (M:true, D:false)
    dtype: bool
    settable: true      # at runtime via callbacks
```

# Ports

Ports describe the inputs and outputs of the block, and can be typed (fixed or templated) or untyped or message ports

```
ports:
-   domain: stream
    id: in
    direction: input
    type: typekeys/T
    multiplicity: '2'

-   domain: stream
    id: out
    direction: output
    type: typekeys/T
```

# Implementations

Specifies implementations/domains for blocks since each block can have multiple variations in the same folder

Normally will be just cpu

```
implementations:
-   id: cpu
# -   id: cuda
```

# Now let's build

meson setup build --prefix=$GR_PREFIX --libdir=lib
cd build && ninja

....   lots of code generation

Taking a look at the auto-generated code in
`build/blocklib/grcon22/multDivSelect`

**Let's see what we get for free ...**

# multDivSelect.h

```
template <class T>
class multDivSelect : virtual public sync_block

{
public:
struct block_args {

        bool select;
        };

    using sptr = std::shared_ptr<multDivSelect>;
    multDivSelect(const block_args& args);
```

**Constructor args lumped together in struct - defaults would be handled here**

```
virtual void set_select(bool select);

virtual bool select();
    protected:
enum params : uint32_t { id_select, num_params };

pmt_sptr param_select;
```

**Setter and getter for our parameter as well as a member PMT object**

**Factory method that will create ptr to desired implementation**

```
enum class available_impl { CPU, PYSHELL };
static sptr make(const block_args& args, available_impl impl = available_impl::CPU);
```

# multDivSelect.cc

```
template<class T> class gr::grcon22::multDivSelect<T>
multDivSelect<T>::multDivSelect(const block_args& args) : sync_block("multDivSelect", "grcon22") {

    for (size_t i = 0; i < 2; i++) {
    add_port(port<T>::make("in" + std::to_string(i),
                          port_direction_t::INPUT,
                          std::vector<size_t>{ 1 }, false));
    }

    for (size_t i = 0; i < 1; i++) {
    add_port(port<T>::make("out" ,
```

**Creation of ports according to yml settings**

```
//d_param_str_map = { {"select", id_select}, };
  d_param_str_map = { {"select", id_select},};
  d_str_param_map = { {id_select, "select"},};

  param_select = std::make_shared<pmtf::pmt>(args.select);

  add_param("select", d_param_str_map["select"], param_select);
```

**Parameter object instantiation and mapping**

```
template <class T>
void multDivSelect<T>::set_select(bool select)
{
    return request_parameter_change(params::id_select, select);
}

template <class T>
bool multDivSelect<T>::select()
{
    return pmtf::get_as<bool>(request_parameter_query(params::id_select));
```

**Setters and Getters wrap base block methods**

**Template instantiations with suffixing**

```
template <>
std::string multDivSelect<std::complex<float>>::suffix(){ return "_cc"; }
template class multDivSelect<std::complex<float>>;
template <>
std::string multDivSelect<float>::suffix(){ return "_ff"; }
template class multDivSelect<float>;
```

# multDivSelect_cpu_gen.h

**Hide some more of the boilerplate**

```
template <class T>
typename multDivSelect<T>::sptr multDivSelect<T>::make_cpu(const block_args& args)
{
    return gnuradio::make_block_sptr<multDivSelect_cpu<T>>(args);
}

template class multDivSelect<std::complex<float>>;
template class multDivSelect<float>;
#define INHERITED_CONSTRUCTORS(type) sync_block("multDivSelect", "grcon22"), multDivSelect<type>(args)
```

# multDivSelect_pybind.cc

This is perhaps the most exciting part for me … free python bindings

```
from gnuradio import grcon22
blk = grcon22.multDivSelect_ff(True)
blk.set_select(False)
```

```
multDivSelect_class.def(py::init([](bool select,  typename gr::grcon22::multDivSelect<T>::available_impl impl) {
                        return ::gr::grcon22::multDivSelect<T>::make({ select }, impl);
                    }),

    py::arg("select"),

                    py::arg("impl") = gr::grcon22::multDivSelect<T>::available_impl::CPU)
    .def_static("make_from_params", &::gr::grcon22::multDivSelect<T>::make_from_params,
    py::arg("json_str"),
    py::arg("impl") = gr::grcon22::multDivSelect<T>::available_impl::CPU)


    .def("set_select", &gr::grcon22::multDivSelect<T>::set_select)

    .def("select", &gr::grcon22::multDivSelect<T>::select)
```

# grcon22_multDivSelect.block.yml

Goal at this point has been to minimally change GRC

Opted for a [4.0 block yml] → [GRC yml] conversion


**We can have a "soft default" used in grc with grc: default: in the parameter**

# The New Block API

The goal up to this point has been to get the block developer to the work() method as quickly as possible, removing roadblocks along the way.

```
template <class T>
multDivSelect_cpu<T>::multDivSelect_cpu(const typename multDivSelect<T>::block_args& args)
    : INHERITED_CONSTRUCTORS(T)
{
}

template <class T>
work_return_t multDivSelect_cpu<T>::work(work_io&)
{
    // Do work specific code here
    return work_return_t::OK;
}
```

**Constructor**

All inputs to work come in this struct ref

**Work Method**

# Work()

## Getting our sample pointers

```
auto in0 = wio.inputs()[0].items<T>(); // can also do ["in0"]
auto in1 = wio.inputs()[1].items<T>();
auto out = wio.outputs()[0].items<T>();

auto noutput_items = wio.outputs()[0].n_items;
```

# Work()

**Getting our block parameter**

– Since the current value of selector lives in the base block as a PMT, we can grab the current value here

```
auto sel = pmtf::get_as<bool>(*this->param_select);
```

**Name matches what we put in the .yml**

For a non-settable parameter, we can just save the value into a private member variable in the constructor

# Work()

## Produce our output samples

```
for (size_t index = 0; index < noutput_items; index++) {
    if (sel) { out[index] = in0[index] * in1[index]; }
    else{ out[index] = in0[index] / in1[index]; }
}

wio.produce_each(noutput_items);
return work_return_t::OK;
```

**Produce/Consume must always be called**

# Write a QA test

Not currently a part of the modtool scripts, but easy to add

1) Create blocklib/grcon22/test/qa_multDivSelect.py (copy from github)
2) Add the test to meson.build

```
test('Mult Div Select', py3, args : files('qa_multDivSelect.py'), env: TEST_ENV)
```

```
ninja

ninja test
```

# Review

1) Created OOT module with script
2) Created block with script
3) Updated .yml file
4) Implemented work function
5) Added QA test
6) Ran example in GRC

# Back to the original vision

# Vision for GNU Radio 4.0

**Modular CPU Runtime**

- Scheduler as plugin
- Application-specific schedulers

**Heterogeneous Architectures**

- Seamless integration of accelerators (e.g., FPGAs, GPUs, DSPs, SoCs)

**Distributed DSP**

- Setup and manage flowgraphs that span multiple nodes

**Straightforward implementation of (distributed) SDR systems that make efficient use of the platform and its accelerators**

# How does this get us to our vision?

In the exercise we really only covered the "straightforward implementation" part of things

Modular CPU Runtime

- Improved CPU scheduler with modular architecture
- Can show performance gains - e.g. by not limiting to TPB

Heterogenous Architectures

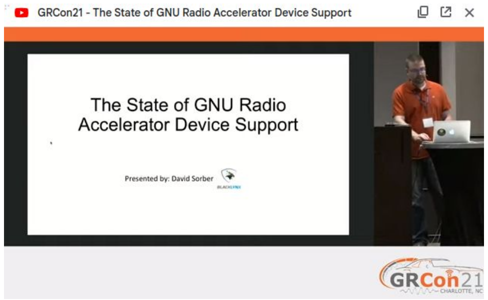
- Custom buffers - take a step beyond 3.10

Distributed DSP

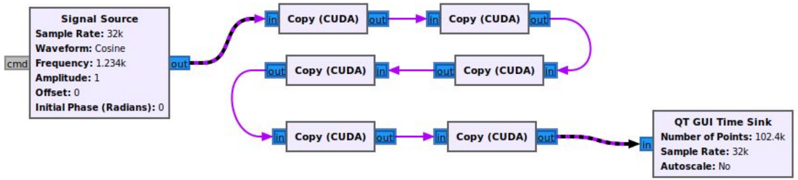
- Because of modular runtime, can create more complex flowgraphs that span multiple compute nodes but controlled from a single node

# Custom Buffers

- 3.10 Feature introduced by **David Sorber** at **Black Lynx** via the **DARPA SDR 4.0** project
  - Final status given last year at GRCon
  - https://www.youtube.com/watch?v=VO1zMXowezg

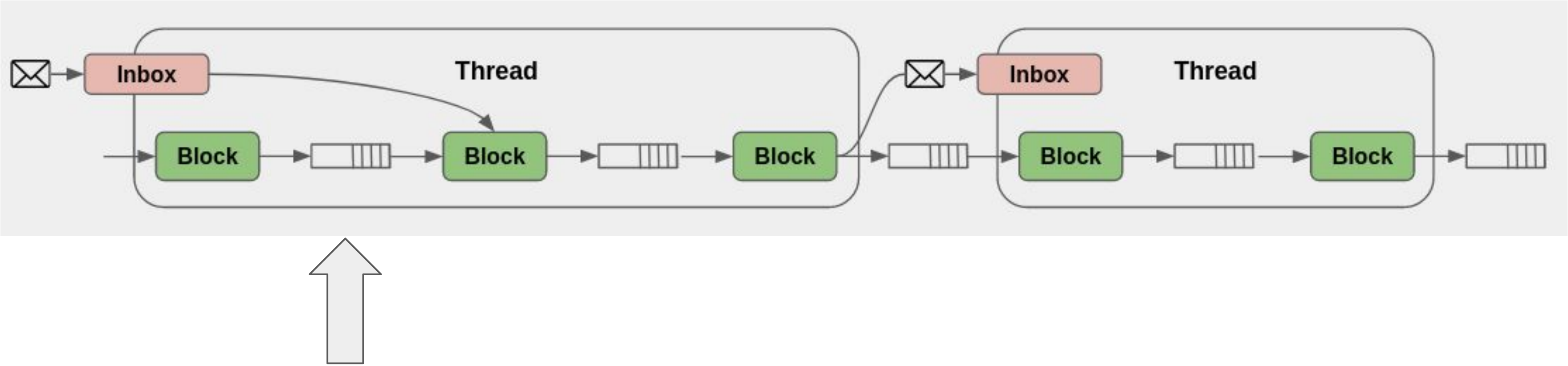



- Device compatible buffer structure (single mapped)
  - https://wiki.gnuradio.org/index.php/Custom_Buffers
- Data able to remain in accelerator memory
  - Streamlined data movement




Prior to 3.10 using custom buffers, each connection between CUDA enabled blocks would require ingress/egress to/from device memory (expensive)

# Custom Buffers




Allow you to specify where the data resides for the buffer that lives in between ports
By default it is the GR double mapped circular buffers (vmcircbuf)

Graphically represented by "domains" in GRC

Bottom Line: In work() we can assume that buffers represent device memory

# Custom Buffers

Some key differences between CB for 4.0

1) NOT built into the block
    a) This was a GR 3.x io_signature API limitation
    b) Assumptions made about ingress/egress that covers most use cases
2) Can specify on each *edge*
    a) More verbose, but more flexible - e.g. different CUDA mem types.

```
tb.connect(src, op).set_custom_buffer(gr.buffer_cuda_properties.make(gr.buffer_cuda_type.H2D))
```

3) Buffer pointer passed into work() via work_io struct
    a) Allows info about the buffer in use to be communicated via the work method that can't be achieved with raw pointers

# Custom Buffers

Need to create derived:

- buffer
- buffer_reader
- buffer_properties

Not going to create a fresh one in this workshop, but we can look at / use:

`buffer_cuda_sm.h`

# Exercise 2: Add CUDA implementation

Prereqs - CUDA installed on your system or via docker + NVIDIA HW

meson configure with enable_cuda

gr built with enable_cuda=true

1) cd build && meson configure ../build -Denable_cuda=true
2) Add CUSP as a subproject
3) uncomment cuda implementation in yml
4) create multDivSelect_cuda.cc and multDivSelect_cuda.h

# multDivSelect_cuda.h

```
#include <cusp/multiply.cuh>
#include <cusp/divide.cuh>
```

**Rather than writing CUDA kernels from scratch, use the CUSP library (homegrown gnuradio volk-like kernel library**

```
private:
    cudaStream_t d_stream;
    std::unique_ptr<cusp::multiply<T>> p_multkernel;
    std::unique_ptr<cusp::divide<T>> p_divkernel;
};
```

# multDivSelect_cuda.cc

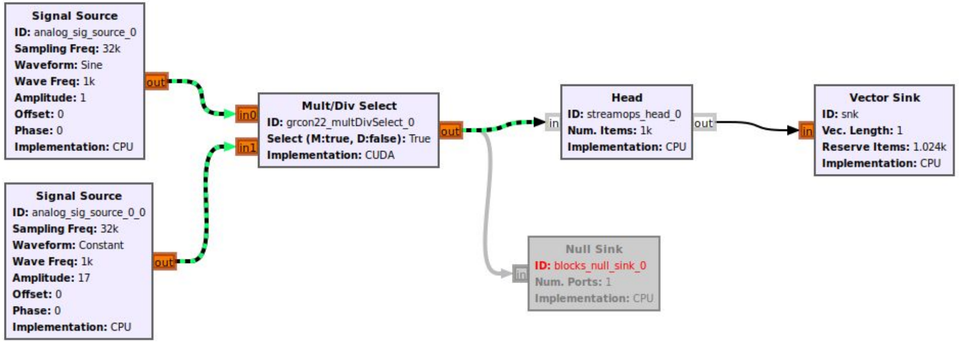
```
template <class T>
multDivSelect_cuda<T>::multDivSelect_cuda(
    const typename multDivSelect<T>::block_args& args)
    : INHERITED_CONSTRUCTORS(T)
{
    p_multkernel = std::make_unique<cusp::multiply<T>>(2);
    p_divkernel = std::make_unique<cusp::divide<T>>(2);

    cudaStreamCreate(&d_stream);
    p_multkernel->set_stream(d_stream);
    p_divkernel->set_stream(d_stream);
}
```

**Block work requires a synchronization as the scheduler expects sample processing to be completed when work returns**

**Good example of where a custom scheduler might increase efficiency**

```
if (sel) {      You, 6 minutes ago • Add cuda impleme
    p_multkernel->launch_default_occupancy(
        {
            { in0, in1 },
        },
        { out },
        noutput_items);
} else {
    p_divkernel->launch_default_occupancy(
        {
            { in0, in1 },
        },
        { out },
        noutput_items);
}

cudaStreamSynchronize(d_stream);

wio.produce_each(noutput_items);
return work_return_t::OK;
```

# Running from GRC




**Switching the "implementation" field in GRC changes the domain and causes rendering to use CUDA implementation and set up custom buffers**

```
Every 0.1s: nvidia-smi

Tue Aug 23 14:46:02 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.65.01    Driver Version: 515.65.01    CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:01:00.0  On |                  N/A |
| N/A   49C    P2    36W /  N/A |   1586MiB /  8192MiB |     21%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      2126      G   /usr/lib/xorg/Xorg                507MiB |
|    0   N/A  N/A      2277      G   /usr/bin/gnome-shell              155MiB |
|    0   N/A  N/A      2744      G   ...AAAAAAAAA= --shared-files       29MiB |
|    0   N/A  N/A      3814      G   ...RendererForSitePerProcess       29MiB |
|    0   N/A  N/A      7085      G   ...0/usr/lib/firefox/firefox      272MiB |
|    0   N/A  N/A      8629      G   ...RendererForSitePerProcess      346MiB |
|    0   N/A  N/A     42856      G   ...624459765613303047,131072      133MiB |
|    0   N/A  N/A    136745      C   /usr/bin/python3                  105MiB |
+-----------------------------------------------------------------------------+
```

# Rendered Flowgraph

```
self.connect((self.analog_sig_source_0, 0), (self.grcon22_multDivSelect_0, 0)).set_custom_buffer(gr.buffer_cuda_properties.make(gr.buffer_cuda_type.H2D))
self.connect((self.analog_sig_source_0_0, 0), (self.grcon22_multDivSelect_0, 1)).set_custom_buffer(gr.buffer_cuda_properties.make(gr.buffer_cuda_type.H2D))
self.connect((self.grcon22_multDivSelect_0, 0), (self.blocks_null_sink_0, 0)).set_custom_buffer(gr.buffer_cuda_properties.make(gr.buffer_cuda_type.D2H))
```

Sets the custom buffer of the generated edge to the desired buffer_properties object

In this case, we have (or GRC has) explicitly specified H2D, D2D, or D2H

```
##################################################
# Blocks
##################################################
self.grcon22_multDivSelect_0 = grcon22.multDivSelect_ff( True, impl=grcon22.multDivSelect_ff.cuda)
```

Also, it's as easy as switching the implementation at instantiation

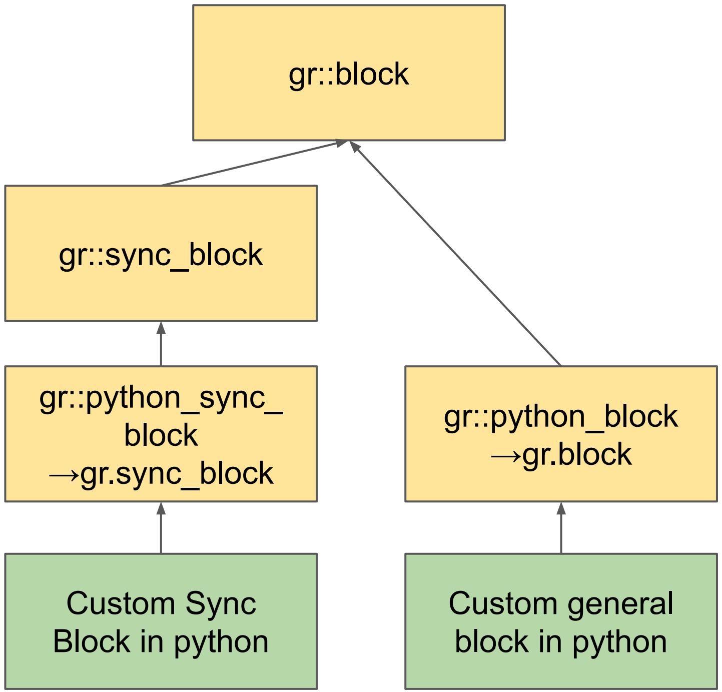
# Exercise 3: Create a Python Block

Let's make the same block again, but implemented in Python

Two mechanisms for creating python blocks:

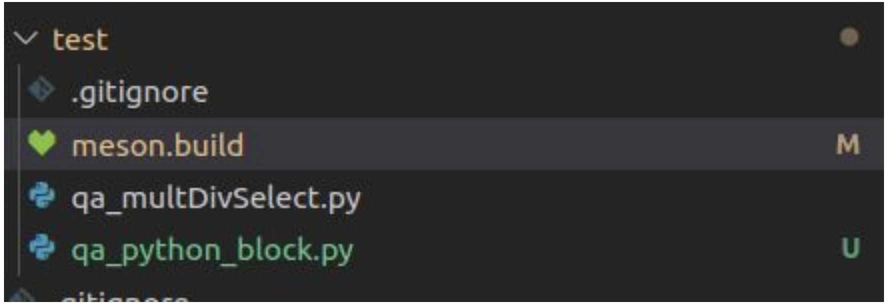
1) Derive from block/sync_block in python_block.h
    a) "from scratch" python block
    b) detached from yaml generation methodology
    c) GRC would have to be manually created
2) Derive from multDivSelect<T>
    a) uses yaml as a starting point
    b) still requires a few manual steps that should be automated

# "From Scratch" python block inheritance



# From scratch python block

Add block directly to a new qa test




```
test('Mult Div Select', py3, args : files('qa_multDivSelect.py'), env: TEST_ENV)
test('Mult Div Select (Python)', py3, args : files('qa_python_block.py'), env: TEST_ENV)
```

# Create the class

```
class multDivSelect_ff(gr.sync_block):
    def __init__(self, select):
        gr.sync_block.__init__(
            self,
            name="multDivSelect")

        self._select = select

        self.add_port_f("in1", gr.INPUT)
        self.add_port_f("in2", gr.INPUT)
        self.add_port_f("out", gr.OUTPUT)

    def work(self, wio):
        noutput_items = wio.outputs()[0].n_items

        inbuf1 = self.get_input_array(wio, 0)
        inbuf2 = self.get_input_array(wio, 1)
        outbuf1 = self.get_output_array(wio, 0)

        if self._select:
            outbuf1[:] = inbuf1 * inbuf2
        else:
            outbuf1[:] = inbuf1 / inbuf2

        wio.produce_each(noutput_items)
        return gr.work_return_t.OK

    # Not thread safe??
    def set_select(self, select):
        self._select = select

    def select(self):
        return self._select
```

**This looks almost exactly like GR 3.X python blocks, except we use add_port instead of io_signature**

**Our setters and getters must be manually specified**

# ... and test

```
def test_mult_f_python(self):
    nsamples = 10000

    indata_1 = list(range(100)) * (nsamples // 100)
    indata_2 = list(range(100)) * (nsamples // 100)

    expected_output = [z[0] * z[1] for z in zip(indata_1, indata_2)]

    src1 = blocks.vector_source_f(indata_1, False)
    src2 = blocks.vector_source_f(indata_2, False)

    blk = multDivSelect_ff(True)
    snk = blocks.vector_sink_f()


    self.fg.connect(src1, 0, blk, 0)
    self.fg.connect(src2, 0, blk, 1)
    self.fg.connect(blk, 0, snk, 0)

    self.fg.start()
    self.fg.wait()

    self.assertSequenceEqual(expected_output, snk.data())
```

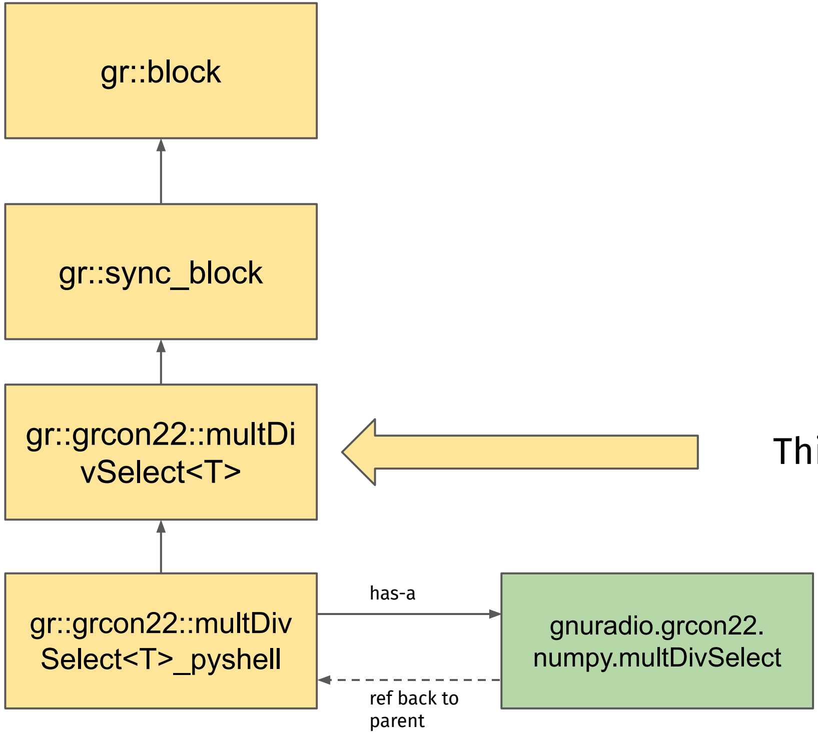
# Extending the existing block

```
implementations:
-   id: cpu
-   id: cuda
-   id: numpy
    lang: python
# -   id: cupy
#     lang: python
#     domain: cuda
```

**Expand our list of implementations to include a "numpy" with language set to python**

**Can also do something similar with a cuda domain implementation in python**

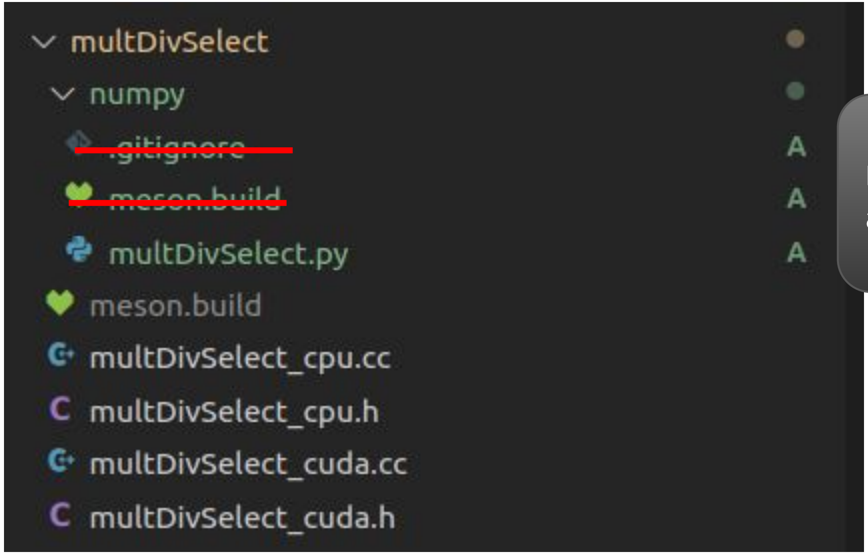
# Extended python block inheritance




This inheritance should give all the 4.0 niceties

# Add the numpy implementation as a directory

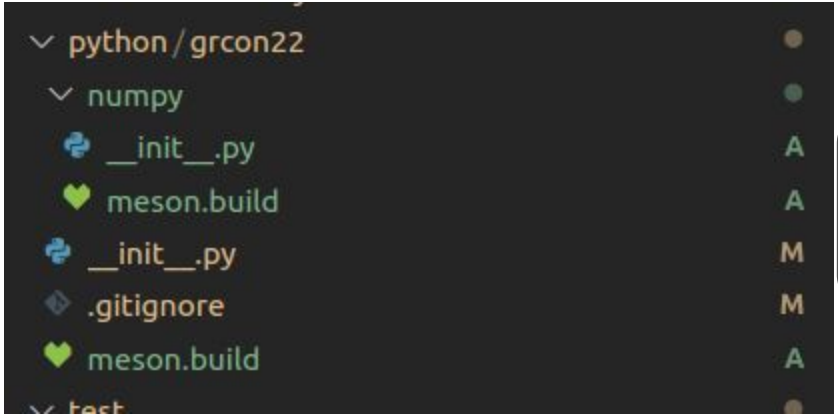
Copy from add.py in the main gnuradio dev-4.0 tree




**meson.build is boilerplate and should be automatically generated**

# Boilerplate

A bit more boilerplate here that also *could* be automated




need to tie in, e.g.
from gnuradio import grcon22.numpy.mulDivSel_ff

# The class extending the base block

```
class multDivSelect_ff():
    def __init__(self, blk, **kwargs):
        self._blk = blk
```

Any constructor parameters from the yaml are passed in as kwargs via the pyshell

blk is a reference back to the pyshell - access to base block methods

*pyshell* is a generic autogenerated shell for any python implementation

# Work()

Same as previous implementation except we now have access to block parameters through self._blk

Convenience methods for getting the numpy arrays from the work_io struct

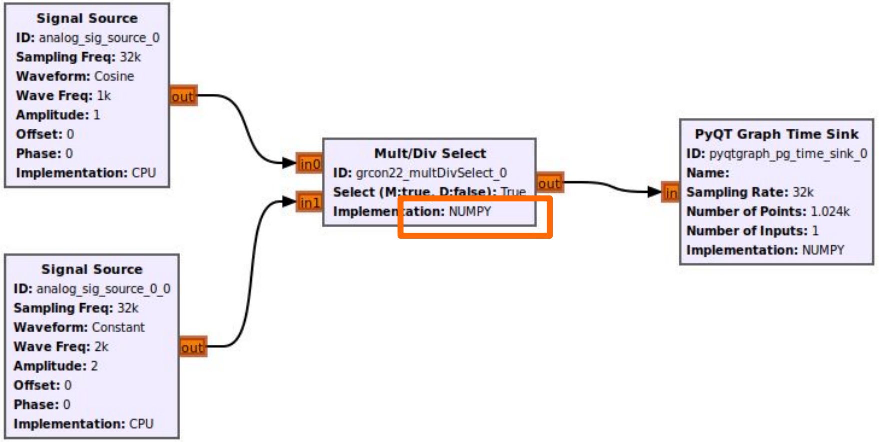
```
def work(self, wio):
    out = wio.outputs()[0]
    noutput_items = out.n_items

    inbuf1 = gr.get_input_array(self._blk, wio, 0)
    inbuf2 = gr.get_input_array(self._blk, wio, 1)
    outbuf1 = gr.get_output_array(self._blk, wio, 0)

    # Get the current value of our parameter
    sel = self._blk.get_parameter("select")
    if sel():  # the __call__ operator gets the native value of the pmt
        outbuf1[:] = inbuf1 * inbuf2
    else:
        outbuf1[:] = inbuf1 / inbuf2

    out.produce(noutput_items)
    return gr.work_return_t.OK        You, last week • Extend multDivSelect
```

# QA test

```python
def test_mult_f_python_extend(self):
    nsamples = 10000

    indata_1 = list(range(100)) * (nsamples // 100)
    indata_2 = list(range(100)) * (nsamples // 100)

    expected_output = [z[0] * z[1] for z in zip(indata_1, indata_2)]

    src1 = blocks.vector_source_f(indata_1, False)
    src2 = blocks.vector_source_f(indata_2, False)

    blk = grcon22.multDivSelect_ff(True, impl=grcon22.multDivSelect_ff.numpy)
    snk = blocks.vector_sink_f()


    self.fg.connect(src1, 0, blk, 0)
    self.fg.connect(src2, 0, blk, 1)
    self.fg.connect(blk, 0, snk, 0)

    self.fg.start()
    self.fg.wait()

    self.assertSequenceEqual(expected_output, snk.data())
```

# GRC

Just set NUMPY as implementation and it will use what we just coded in python



# Additional Block API Considerations

# Forecasting

The GR3 forecasting mechanism is useful for informing the scheduler the appropriate buffer sizes to provide

It is however locked into a singular scheduling paradigm (backpressure based). (See tagged stream blocks - forward output to input calculation)

Since scheduling is becoming modular, we want to be flexible in the mechanism exposed from the block API to the scheduler

Instead, return from the work function if the provided buffers are not sufficient

Still open discussion on whether a `check_work` method would be appropriate

# History

History() in GR3 is a useful feature for a subset of blocks that maintain access to the previous N-1 samples

However, it overly complicates the scheduler, and since it only affects a small percentage of blocks, we can deal with it in the block itself

tl;dr - don't consume all the samples

# Forecasting/History in fir_filter block

We ensure that the output provided is greater than the input plus the internally maintained history variable

`noutput_items` will be less than the available inputs

But we only consume `noutput_items` on both input and output

```
template <class IN_T, class OUT_T, class TAP_T>
work_return_t fir_filter_cpu<IN_T, OUT_T, TAP_T>::work(work_io& wio)
{
    // Do forecasting
    size_t ninput = wio.inputs()[0].n_items;
    size_t noutput = wio.outputs()[0].n_items;

    auto decim = pmtf::get_as<size_t>(*this->param_decimation);

    if (d_updated) {
        d_hist_change = d_history - d_fir.ntaps();
        d_history = d_fir.ntaps();
        d_updated = false;
        d_hist_updated = true;
    }

    auto min_ninput = std::min(noutput * decim + d_history - 1, ninput - (d_history - 1));
    // auto noutput_items = std::min( (min_ninput + decim - 1) / decim, noutput);
    auto noutput_items = std::min(min_ninput / decim, noutput);

    if (noutput_items <= 0) {
        return work_return_t::INSUFFICIENT_INPUT_ITEMS;
    }
```

# Message Ports

Every parameter can be updated through `param_update` message port - get this for free

```
block::block(const std::string& name, const std::string& module)
    : node(name),
      s_module(module),
      d_tag_propagation_policy(tag_propagation_policy_t::TPP_ALL_TO_ALL)
{
    // {# add message handler port for parameter updates#}
    auto msg_param_update = message_port::make("param_update", port_direction_t::INPUT);
    msg_param_update->register_callback(
        [this](pmtf::pmt msg) { this->handle_msg_param_update(msg); });
    add_port(std::move(msg_param_update));

    auto msg_system = message_port::make("system", port_direction_t::INPUT);
    msg_system->register_callback(       mormj, last month • global: Initial port of newsch
        [this](pmtf::pmt msg) { this->handle_msg_system(msg); });
    add_port(std::move(msg_system));
}
```

```
void block::handle_msg_param_update(pmtf::pmt msg)
{
    // Update messages are a pmtf::map with the name of
    // the param as the "id" field, and the pmt::wrap
    // that holds the update as the "value" field

    auto id = pmtf::string(pmtf::map(msg)["id"]).data();
    auto value = pmtf::map(msg)["value"];

    request_parameter_change(get_param_id(id), value, false);
}
```

# Message Ports

For custom message ports, driven through yaml workflow

```yaml
ports:
-   domain: message
    id: print
    direction: input
    optional: true
-   domain: message
    id: store
    direction: input
    optional: true
```

**With a message port defined the yaml, block will expect a handle_{id} for each block implementation**

```cpp
private:
    std::vector<pmtf::pmt> d_messages;
    void handle_msg_print(pmtf::pmt msg) override;
    void handle_msg_store(pmtf::pmt msg) override;
};
```

# Message Port Performance

<insert graph showing benchmarking>

Makes PDU based flowgraph much more feasible

In the scheduler, uses the same mechanism as for stream buffer updates

Part of speedup is reduced reliance on PMT identifiers, part is improved PMT design

# Creating Blocks "in-tree"

`--intree` flag with `create_block.py` script

e.g. to create a block in analog

cd blocklib/analog

create_block.py ... --intree

# Moving Forward -
# getting to a solid 4.0.0

# Moving Forward

Biggest missing items:

- Visualization Blocks
  - e.g. qtgui refresh/replacement
- Radio Blocks (Soapy/UHD/IIO)
  - Not that hard but want to keep generic/consistent
  - https://github.com/gnuradio/gnuradio/pull/6028
- Documentation
  - Since c++ .h files not the primary entrypoint, need another solution
  - Tied in with .yaml and organized →readthedocs.io or something
- Begin Port Block Library
  - If everyone is happy with current API …

# Moving Forward

Items that require fixing/revisiting

- Generalized Callbacks tied in with yaml
- Evaluate dependencies
- Better GRC Integration
    - Move to QT GRC?
- ...