GR 4.0 Workshop

GNU Radio Conference 2021

Josh Morman
Bastian Bloessl
Garrett Vanhoy

Purpose

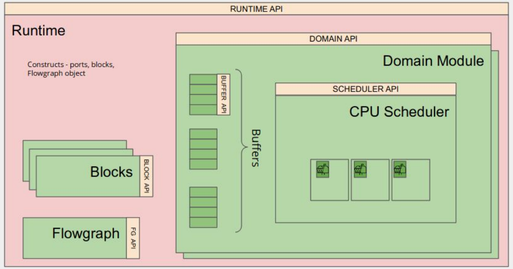
- Step through the proposed changes for GR 4.0 (newsched)
- Facilitate discussion about ongoing design decisions
- Get a hands-on feel for what core and application development may be like in a future GR
- Prioritize development items for the near and medium term

Feedback and comments are very much appreciated

Overview

At GRCON20, we presented a high level overview of the proposed runtime

In the meantime, we have been able to continue development on the proof of concept (newsched) and learn what is feasible, what is not, and bring in some new concepts



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

Agenda

- Overview
- Block Interfaces
  - Major changes to the Block API
- Block generation process
  - YAML driven block design to create and maintain blocks with less steps
- Scheduler Interfaces
  - How modular schedulers interact with the runtime
- Custom Buffers
  - What is different from the SDR 4.0 work
- Benchmarking
  - How does GR 4.0 improve the performance state of things - some examples

Getting Set Up

newsched repo:

- https://github.com/gnuradio/newsched

Docker either for reference, or sandbox

- https://github.com/mormj/newsched-docker/

also, can
docker pull mormj/newsched-demo

Newsched can be built (might need additional steps if using CUDA)

1. Create a prefix
   a. mkdir /path/to/prefix/src
   b. cd /path/to/prefix/src
   c. copy setup_env.sh from newsched-docker
   d. git clone https://github.com/gnuradio/newsched
   e. cd newsched

```
meson setup build
cd build
ninja
```

```
(adjust newsched-docker as needed)
```

Why meson?

https://mesonbuild.com/

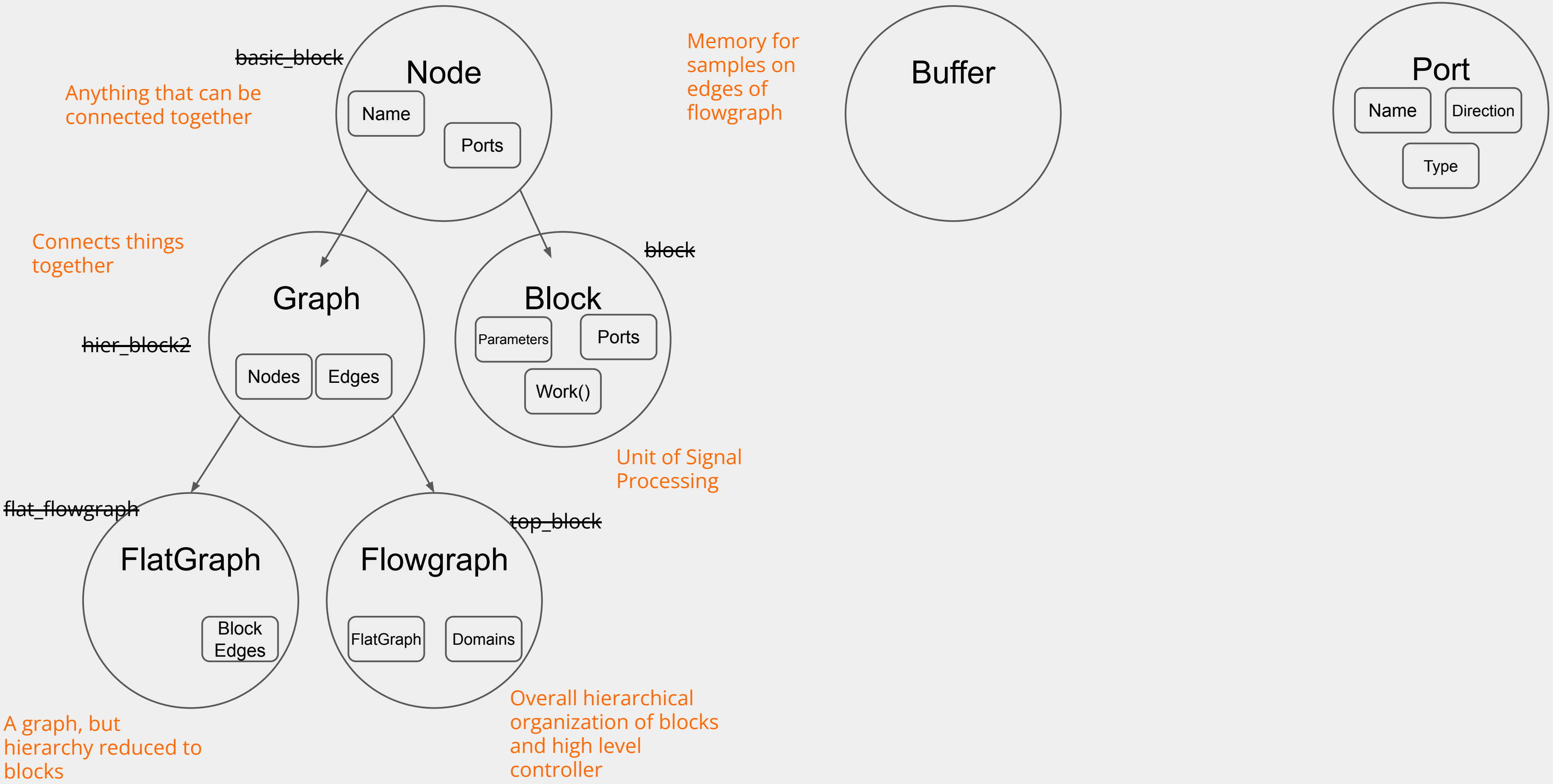
Tradeoff between getting things working quickly and the flexibility of a more powerful build system (e.g. CMake)

I have stuck with the "quickly" aspect of things

Will do a port to CMake later on

- Lack of macros/functions means a lot of duplicated code
- But lack of macros is part of the meson design philosophy

Class Inheritance Hierarchy



Block API

- The big changes - *simplify, simplify, simplify*
  - no more **forecast**() - return the same info in the work()
  - no more **history**() - overcomplicates schedulers
  - work() directly callable - allow for "scheduler-less" operation
  - work() takes input/output structs - more flexibility
  - no more dynamic nports - blocks will need parameter to specify nports
  - **port** as first-class object
  - multiple implementations per block - e.g. CPU, CUDA, OpenCL, ...
  - auto-generated code - less places to type things, auto pybind
- TBD
  - other scheduler hints - *output_multiple*, *relative_rate*, sample_delay, etc.
- The future
  - **Parameters** as centralized path for setters/getters, messages, tags, RPC
  - even more auto-generation, better tools

Block API - work()

```
   virtual work_return_code_t work(std::vector<block_work_input>& work_input,
                                   std::vector<block_work_output>& work_output)
```

```
struct block_work_input {
   int n_items;
   buffer_reader_sptr buffer;
   int n_consumed; // output the number of items that were consumed on the work() call
```

```
struct block_work_output {
   int n_items;
   buffer_sptr buffer;
   int n_produced; // output the number of items that were produced on the work() call
```

Block API - work()

forecast() - how to handle lack thereof

at the beginning of work, set your condition

```
    if (work_input[0].n_items < (int) d_length)
    {
        work_output[0].n_produced = 0;
        work_input[0].n_consumed = 0;
        return work_return_code_t::WORK_INSUFFICIENT_INPUT_ITEMS;
    }
```

Also, can add more fields to indicate more information to scheduler:

- Insufficient, but how many did I need?
- Don't call again for an amount time

Block API - work()

history() - how to handle lack thereof

Simple - just don't consume all the samples

- class variable d_history
- "forecast" d_history greater than you will consume
- don't consume d_history samples

Motivation - the number of blocks that use history is limited, but it puts a substantial burden on the corner cases for schedulers

Schedulers will still have to solve the problem of not all samples being consumed

Block API - port

Ports are

- typed
- untyped
- message

Handle some of things that GRC handles at the higher layer

- multiplicity

Ports can be connected together with items of same size

- Is there a need for e.g. stream_to_vector
- Connect different sized objects together
    - creates nightmares with tags to be figured out

"Scheduler-less" Operation

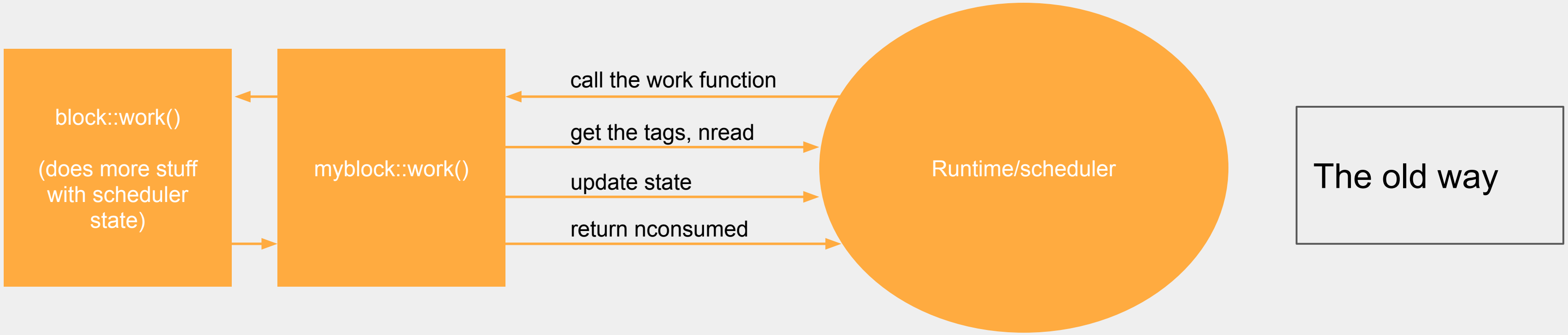
Why Scheduler-less

Why would we want to call a block's work function without a scheduler

- For debug
- For QA/validation of the work function
- For offline simulation scripts for prototyping (i.e. MATLAB -like)
  - but still use the same dsp as GNU Radio
- Push data through rather than stream/schedule

Why callable work() is not currently possible

- In Current GR, the scheduler API is all jumbled up with the block API, e.g. a typical work() function where the signal processing in the block happens
- Scheduler calls work function, work function calls back into scheduler – not clean
- (one goal) To make work() directly callable, we need to keep the block API clean
- Currently in newsched, the exception to this is message ports
  - post() to a port within a block will immediately put the message via its parent interfaces onto the queue of a scheduler



What needs to happen for scheduler-less

1. **Done**
    a. De-couple scheduling from work()

2. **[in-progress]**
    a. Python [pybind] bindings to convert:
        i. numpy arrays ← → gr::gr_block_work_io objects
        ii. Python dicts ← → gr::tag_t objects
    b. Error handling associated with conversions
    c. A wrapper in C++ around the work() function that only takes in buffers for inputs and allocates buffers for outputs (much like MATLAB's step)

An Early Example

```
mult = blocks.multiply_const_ff(1.0, 1)

work_input = gr.block_work_input(np.ones((10,)))
work_output = gr.block_work_output(np.zeros((10,)))

mult.work([work_input], [work_output])

input_vec = work_input.buffer.numpy()
output_vec = work_output.buffer.numpy()
```

Where is this being done?

gnuradio/newsched: *branch* gvanhoy/direct_block_interface

Files: newched/runtime/python/gr/bindings

Block Design Workflow

YAML Driven Block Workflow

With multiple implementations per block being added, organization of the code becomes key

Problem:

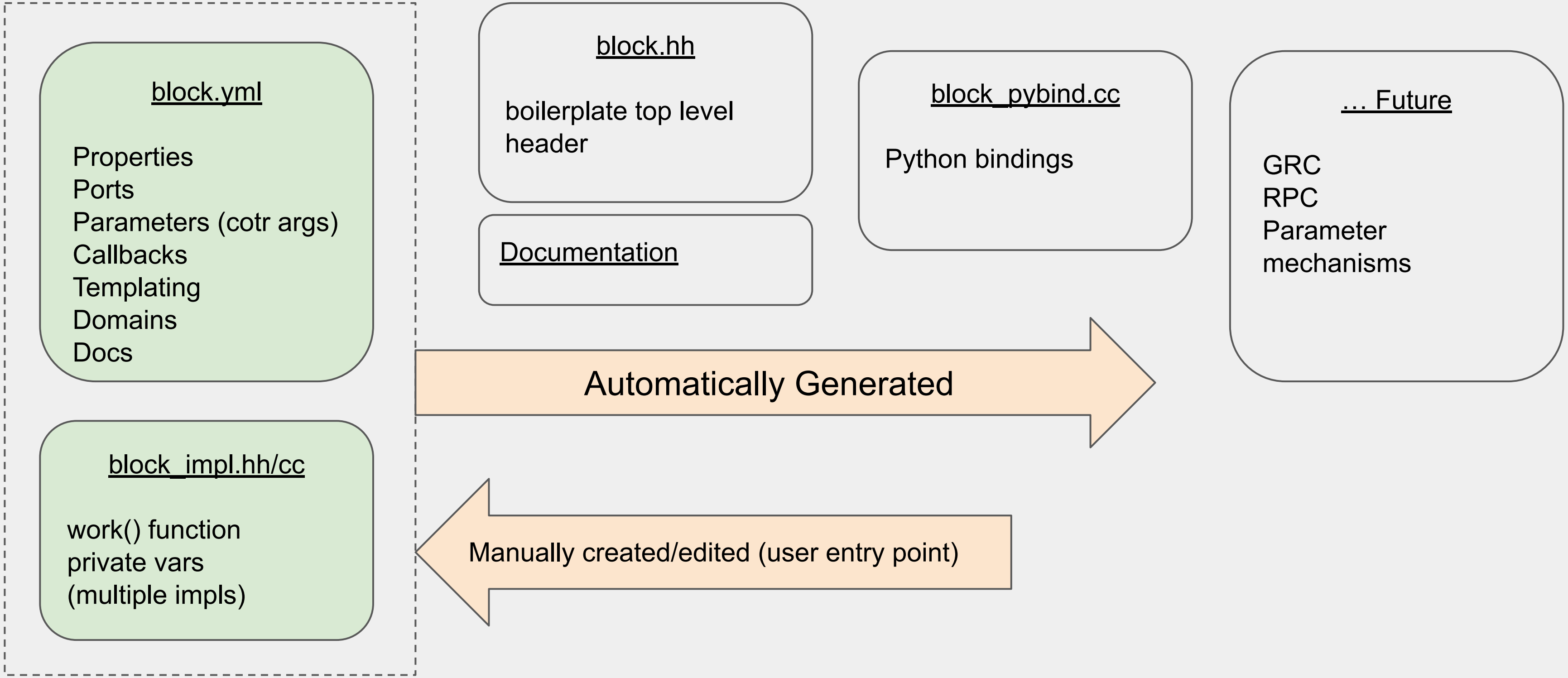
- Currently there is a lot of boilerplate that the user has to do manually (after modtool is done) - e.g. add a parameter
- This becomes a barrier to people creating usable DSP in GNU Radio
- With multiple implementations per block (CUDA, openCL, XRT, …), the block library grows in size and complexity

Goals:

- multiple implementations
- get the user to work() function quicker
- minimize boilerplate through automation
- unify interfaces/mechanisms (constructor, setters, tags, RPC, messages) via automation

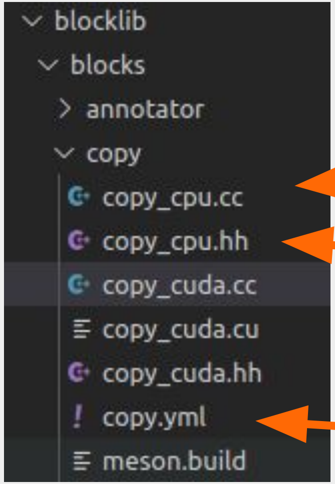
GR 4.0 - Block Creation Workflow

YAML Driven Architecture



Basic File Structure

Manually managed files



3) c++ file for the implementation

2) header file for the implementation

1) YAML file describes block

Example Files

```yaml
module: blocks
block: copy
label: Copy

properties:
-   id: blocktype
    value: sync

parameters:
-   id: itemsize
    label: Item Size
    dtype: size_t
    settable: false

ports:
-   domain: stream
    id: in
    direction: input
    type: untyped
    size: itemsize

-   domain: stream
    id: out
    direction: output
    type: untyped
    size: itemsize

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

You, 2 weeks ago | 1 author (You)

```cpp
class copy_cpu : public copy
{
public:
    copy_cpu(block_args args) : copy(args), d_itemsize(args.itemsize) {}
    virtual work_return_code_t work(std::vector<block_work_input>& work_input,
                                    std::vector<block_work_output>& work_output) override;

protected:
    size_t d_itemsize;
};
```

```cpp
copy::sptr copy::make_cpu(const block_args& args) { return std::make_shared<copy_cpu>(args); }

work_return_code_t copy_cpu::work(std::vector<block_work_input>& work_input,
                                  std::vector<block_work_output>& work_output)
{
    auto* iptr = (uint8_t*)work_input[0].items();
    int size = work_output[0].n_items * d_itemsize;
    auto* optr = (uint8_t*)work_output[0].items();
    // std::copy(iptr, iptr + size, optr);
    memcpy(optr, iptr, size);

    work_output[0].n_produced = work_output[0].n_items;
    return work_return_code_t::WORK_OK;
}
```

Templated Example

```yaml
module: math
block: multiply_const
label: Multiply Constant

properties:
-   id: blocktype
    value: sync
-   id: templates
    keys:
    - id: T
      type: class
      options:
        - value: int16_t
          suffix: ss
        - value: int32_t
          suffix: ii
        - value: float
          suffix: ff
        - value: gr_complex
          suffix: cc

parameters:
-   id: k
    label: Constant
    dtype: T
    settable: true
-   id: vlen
    label: Vec. Length
    dtype: size_t
    settable: false
    default: 1
```

Goal to have more templating in blocks to encapsulate common code

e.g. in gnuradio, separate implementations for blocks that do the same thing for float, complex, short

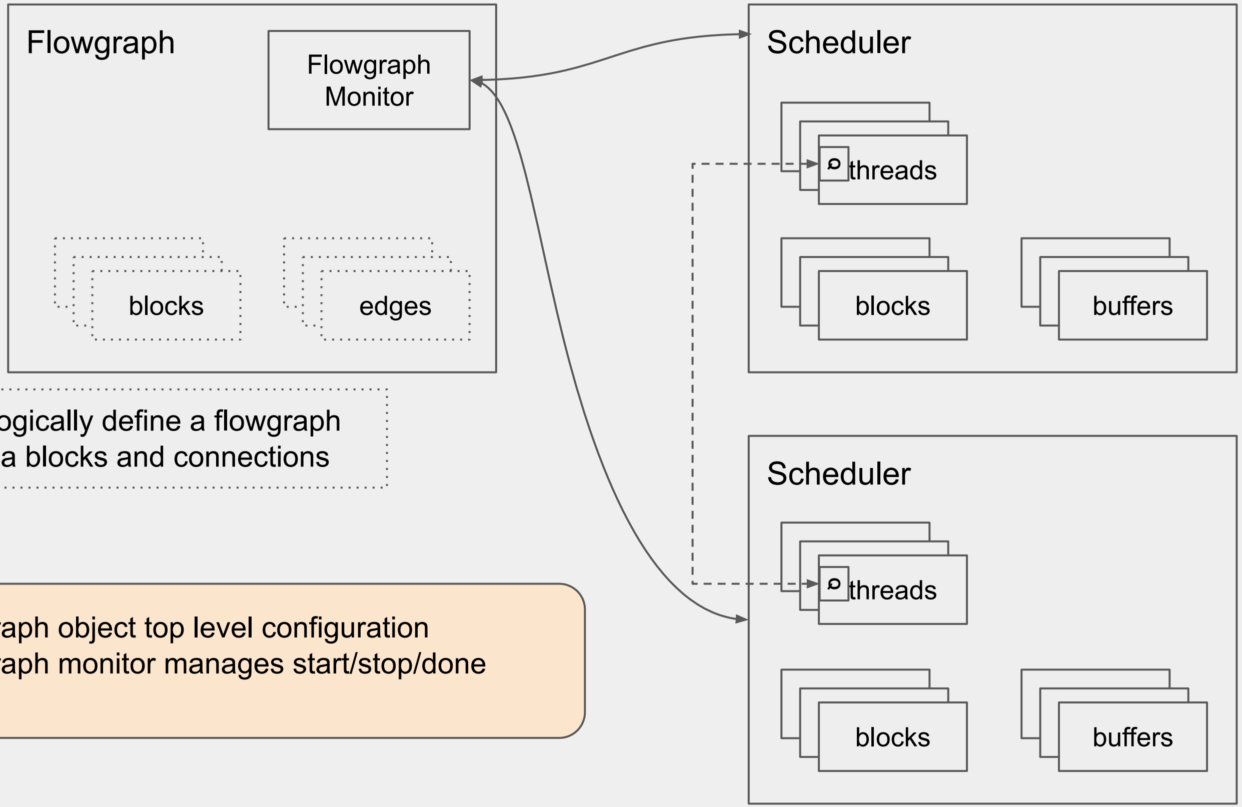
```cpp
template <class T>
typename multiply_const<T>::sptr multiply_const<T>::make_cpu(const block_args& args)
{
    return std::make_shared<multiply_const_cpu<T>>(args);
}

template <class T>
multiply_const_cpu<T>::multiply_const_cpu(const typename multiply_const<T>::block_args& args)
    : multiply_const<T>(args), d_k(args.k), d_vlen(args.vlen)
{
}

template <>
work_return_code_t
multiply_const_cpu<float>::work(std::vector<block_work_input>& work_input,
                                std::vector<block_work_output>& work_output)
{
```

Scheduler Design

Structure and Terminology



Logically define a flowgraph
via blocks and connections

Flowgraph object top level configuration
Flowgraph monitor manages start/stop/done

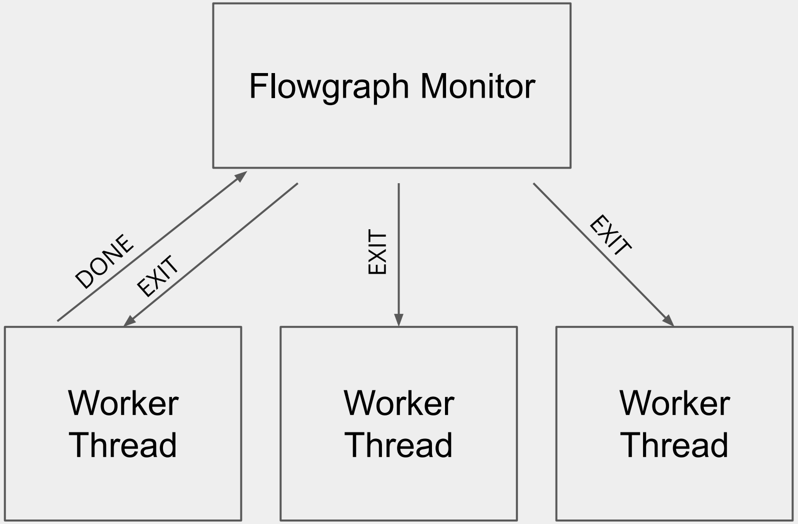
Flowgraph Monitor

Top level object in the runtime to monitor flowgraph execution

Now that execution is potentially spread across multiple schedulers

The entity that can get a response that a block has finished and tell the rest of the blocks to finish as well

Could also be used for flowgraph introspection in a distributed case



Scheduler Interface

A big part of the design is having modularity since we can't solve the scheduling problem for all architectures and applications

Currently, this is the interface:

```
virtual void initialize(flat_graph_sptr fg, flowgraph_monitor_sptr fgmon) = 0;
-   // Instruct the scheduler to initialize buffers, threads, etc.
virtual void push_message(scheduler_message_sptr msg) = 0;
-   // Push a message onto the input queue (or distribute to worker threads)
```

```
virtual void start() = 0;
virtual void stop() = 0;
virtual void wait() = 0;
```

Creating Your Own Scheduler

scheduler_mysched.hh/cc implements scheduler interface

- `push_message`() - need some sort of queue so this method can return right away
- `initialize`() - launch the thread(s) to service the queue, create buffers for the edges in the flowgraph, store the flowgraph objects
- `start/stop` - at least pass the start/stop messages to the blocks
- plugin factory interface - currently half-baked

Scheduler Messages - scheduler_message.hh

Common base class for all messages going into the

```
enum class scheduler_message_t {
    SCHEDULER_ACTION,
    MSGPORT_MESSAGE,
};
class scheduler_message
```

For Scheduler Actions (notify the scheduler of some event such as data ready)

```
enum class scheduler_action_t { DONE, NOTIFY_OUTPUT, NOTIFY_INPUT, NOTIFY_ALL, EXIT };
class scheduler_action : public scheduler_message
```

For Messages, use a different type containing a callback

```
class msgport_message : public scheduler_message
```

N-Block/Thread (NBT) Scheduler

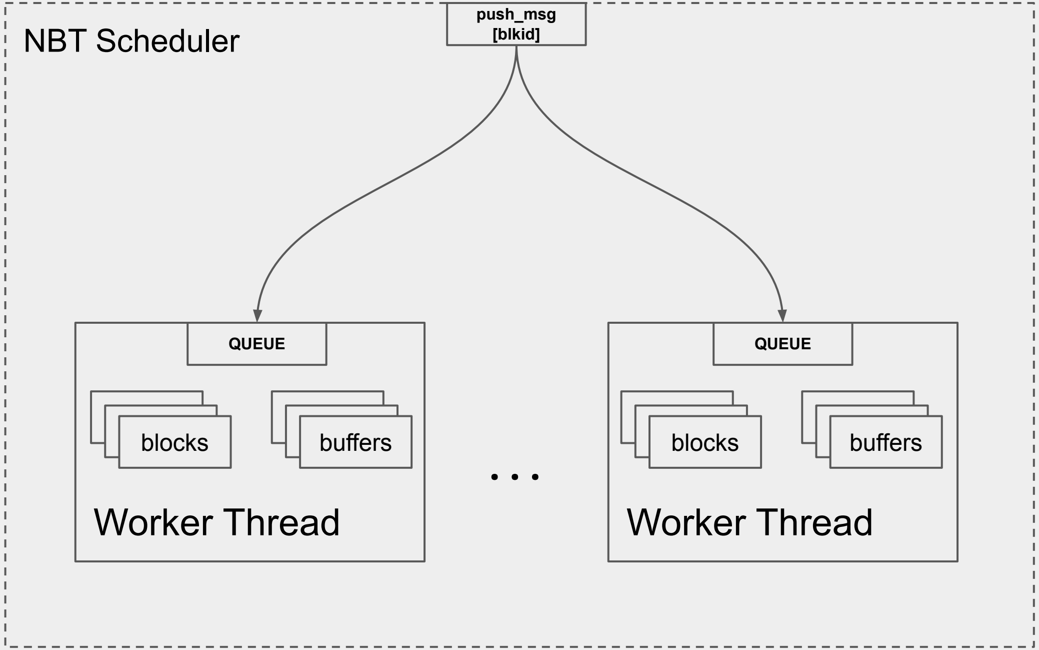
Defaults to TPB

- Unless add_block_group (vector<block_sptr>) is called

Thread blocks on Queue

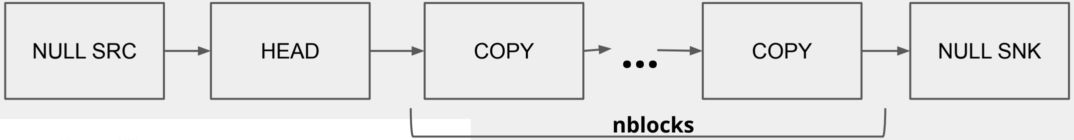
If message is available, acts accordingly

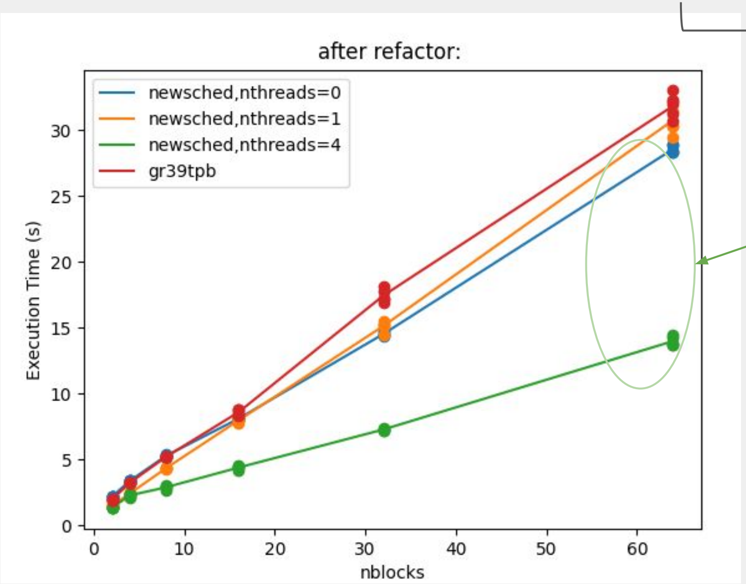
Meat of scheduler in \`graph_executor.cc\`



Scheduler Benchmarks

Following methodology from gr-sched and associated paper





The thread grouping has the intended effect

For newsched:

- nthreads=0 => TPB
- nthreads=4, blocks grouped sequentially in nblocks/nthreads with the src, snk, head joining the adjacent block groups

NBT Scheduler - Thread Wrapper

1. Block on input queue
2. Decode the message
3. Handle accordingly

    NOTIFY_{INPUT,OUTPUT}
    - Cause run_one_iteration to be called

    DONE
    - Signal that a block requested flowgraph done, flush buffers and then notify FGM

    EXIT
    - Immediately exit the thread (FGM signaled flowgraph completion

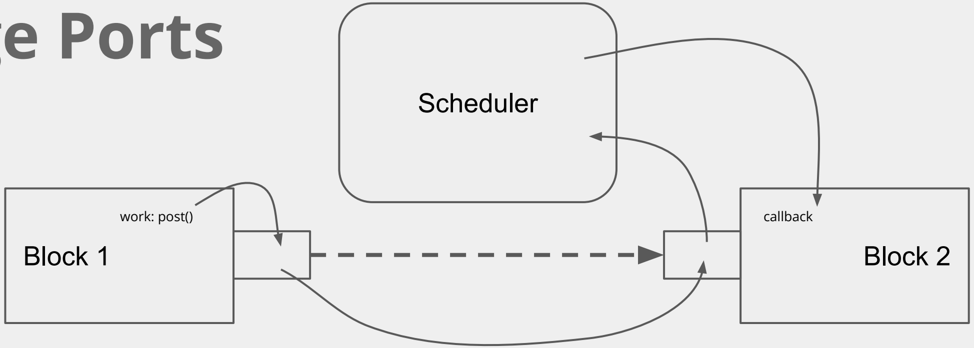
    MESSAGE
    - Call the callback() method

NBT Scheduler - Executor - graph_executor.cc

very similar to GR Block Executor - more(over) simplified

```
run_one_iteration  // (someone told me i needed to do some work)
{
    foreach (b: blocks) ← // TODO intelligently decide the order of blocks to schedule
    {
        foreach (p: b.ports())
        {
            // how much buffer space available
            // prepare work_{input,output}
        }
        {
            b.do_work()
            // adjust buffers, try again if necessary
            // update tags
            // update buffer pointers
        }
    }
    return status
}
```

Message Ports



When connect() takes place between blocks/ports on a graph

- downstream port given reference to upstream port object
  - connected_ports()
- Scheduler also responsible for informing ports of their "parent interface"

From inside a work() function, post(pmt) to the port object

- Receiving port will pass the message ptr to its owning scheduler, and get placed on the queue

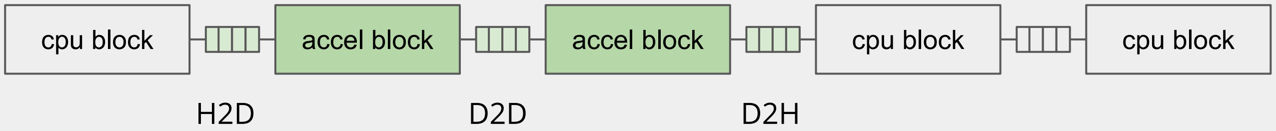
Custom Buffers

Interface

Buffer is associated with edge in graph

Assumption: in work(), in and out buffers are ***already in appropriate device memory*** - e.g. should not have H2D or D2H memcpy in work()

Depending on placement of accelerated block, custom buffers need to be on both upstream and downstream edge



Abstract Buffer API - buffer.hpp

```
public:

    virtual bool read_info(buffer_info_t &info) = 0;
    virtual bool write_info(buffer_info_t &info) = 0;
```

Return the state of the read/write buffer

Looks much like current GR buffer API

```
    virtual void post_read(int num_items) = 0;
    virtual void post_write(int num_items) = 0;
```

Tell the buffer what was done to it, so it can update pointers

Interface


```
flowgraph->connect(src,blk1)->set_buffer(CUDA_BUFFER_ARGS_H2D)
flowgraph->connect(blk1,blk2)->set_buffer(CUDA_BUFFER_ARGS_D2D)
flowgraph->connect(blk2,blk3)->set_buffer(CUDA_BUFFER_ARGS_D2H)
flowgraph->connect(blk3,snk) // uses default buffer

flowgraph->run()
```

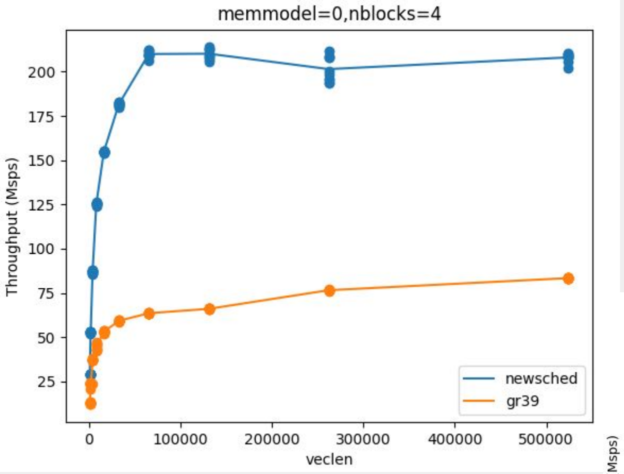
Custom Buffer Benchmarks

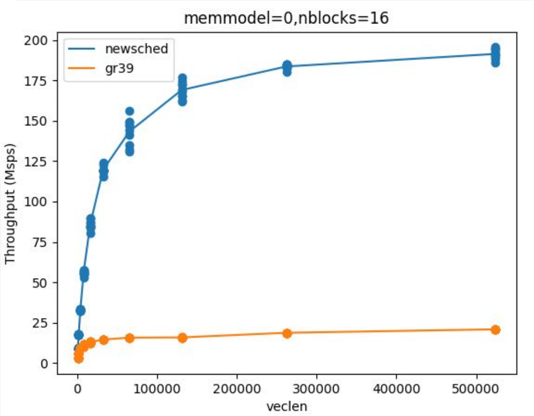
Benefit decreases as kernel execution time increases

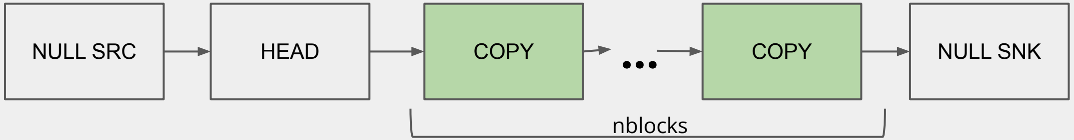
memmodel 0: H2D, D2D, D2H
veclen is batch_size into gpu

In the gr39 case, the H2D, D2H is done in every work() call

In the newsched case, custom buffers call the work() function assuming data is already accessible by gpu (either in device or pinned memory)







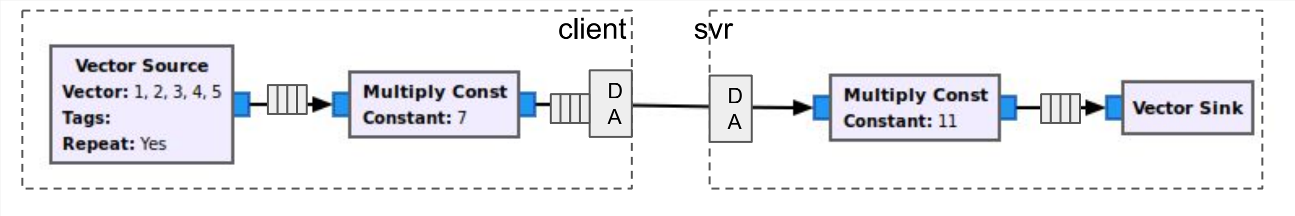
these copy blocks running on gpu accelerator

Good Bye Domain Adapters

Domain Adapters were an attempt to abstract buffers over a connection between blocks handled by different schedulers (potentially on different compute nodes)

Became difficult to handle cleanly in the scheduler, and not well thought out enough to apply to distributed flowgraphs

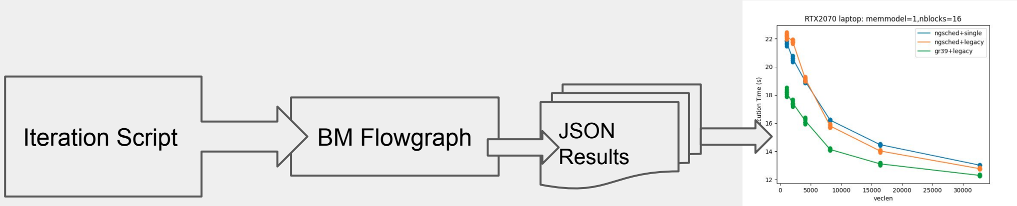
Concept still needs to be ironed out, but not necessary *for now*



Benchmarking

Benchmarking/Profiling Tools

- gr-bench (based on gr-sched) (https://github.com/mormj/gr-bench)
  - Some python scripts for iterating benchmark flowgraphs and plotting results



- nvprof/nvvp
  - For profiling CUDA applications, shows/traces relative time spent by memcpy, kernel launch executions
- prof/flamegraph-rs
  - For non-CUDA applications, sampling profiler to show proportion of execution time spent in each function

Benchmarking

Primarily interested in flowgraph execution time

Method: Create a parameterized flowgraph that prints to stdout:
[PROFILE_TIME] time_in_sec [PROFILE_TIME]

Gather these up over a range of parameters and plot

Example ...

TODO: The Future

What capabilities do we want to expand in the future

- GRC/modtool integration
- Blocking I/O
- Distributed Operation
- CMake or figure out meson
- Implementation extensions of in-tree blocks
  - (don't have CUDA in-tree)
- Async Scheduler/Runtime
- Commonality between parameter access mechanisms
  - first attempt was here: https://github.com/gnuradio/newsched/pull/71

Parameter Access Mechanisms

The current mechanism for having publicly exposed variables requires a lot of manual code intervention

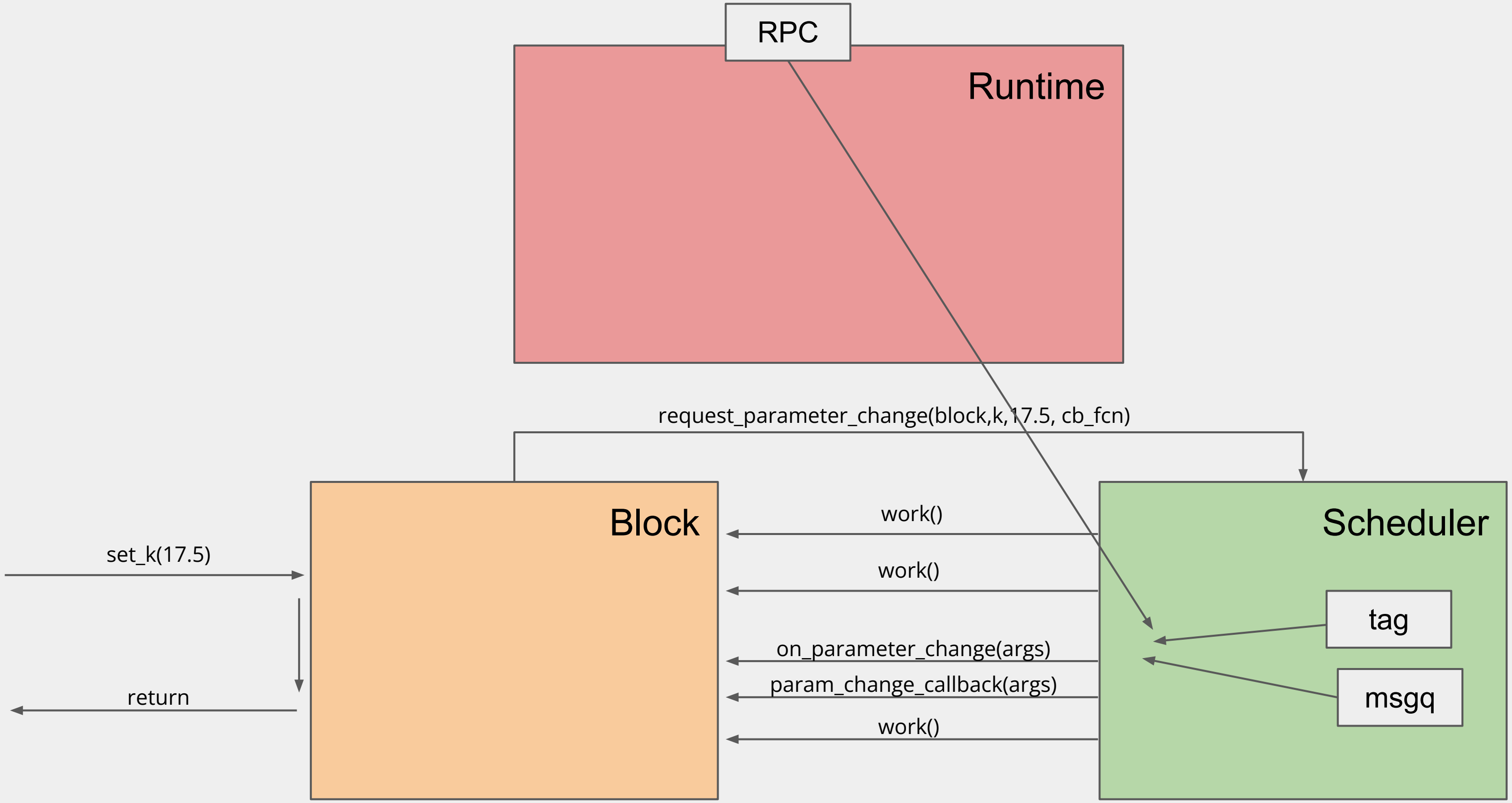
- constructor args
- setters/getters
- message ports
- RPC
- tags

Define each parameter once, then wrap changes in the other routes to change parameters generically

Also, parameters can be scheduled to change at a particular sample number if they pass through the scheduler

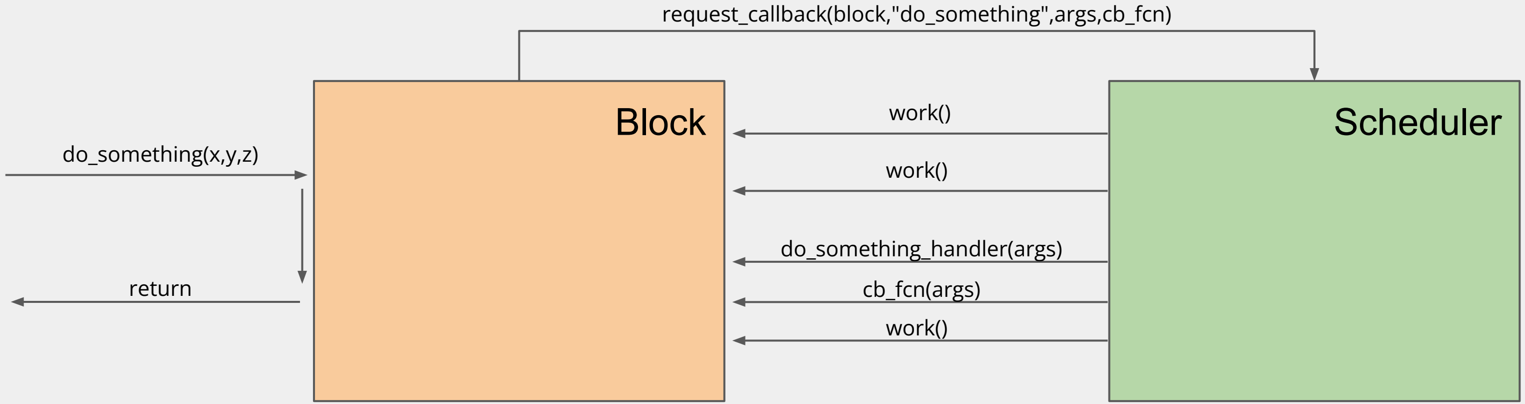
Reduce the burden of making parameter changes threadsafe by ensuring callbacks don't collide with the work function by passing through the scheduler

Parameters


RPC
Runtime
request_parameter_change(block,k,17.5, cb_fcn)
Block
Scheduler
set_k(17.5)
return
work()
work()
on_parameter_change(args)
param_change_callback(args)
work()
tag
msgq

Callbacks

Like parameter changes, general public functions need to pass through the scheduler to remove the thread safety requirement on the block



Same path holds for RPC, message port, etc

Tutorial

Getting Set Up

newsched repo:

- https://github.com/gnuradio/newsched

Docker either for reference, or sandbox

- https://github.com/mormj/newsched-docker/

Newsched can be built (might need additional steps if using CUDA)

1. Create a prefix
   a. mkdir /path/to/prefix/src
   b. cd /path/to/prefix/src
   c. copy setup_env.sh from newsched-docker
   d. git clone https://github.com/gnuradio/newsched
   e. cd newsched

```
meson setup build
cd build
ninja
```

```
(adjust newsched-docker as needed)
```

docker run --network=host -it --rm -v `pwd`:/workspace/code newsched-demo-nocuda bash

Docker with CUDA

https://nvidia.github.io/nvidia-docker/

- Set up the repository

apt install nvidia-container-toolkit

docker run --network=host --gpus all -it --rm -v `pwd`:/workspace/code newsched-demo bash