Yet another new SDR runtime?

Dr. Daniel Estévez

1 February 2025
FOSDEM, Brussels

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

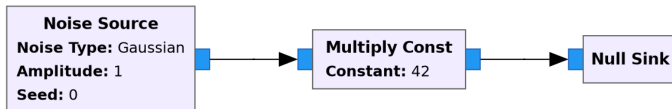
Asking the big questions

- Which is the *one true best* SDR runtime?

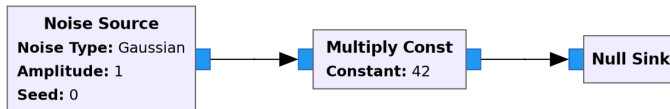
Asking the big questions

- Which is the *one true best* SDR runtime?
- What makes an SDR runtime be fast?
- Look at GNU Radio 3.10, GNU Radio 4.0, and FutureSDR

All the SDR runtimes look the same!

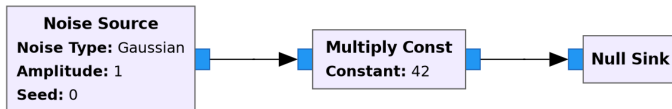


All the SDR runtimes look the same!



- Connections in the flowgraph are a circular buffer shared between the connected blocks (single-producer multi-consumer)

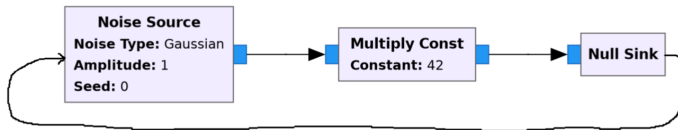
All the SDR runtimes look the same!



- Connections in the flowgraph are a circular buffer shared between the connected blocks (single-producer multi-consumer)
- Different input and output buffers. Bad for cache.
- High (and difficult to control) latency on TX flowgraphs

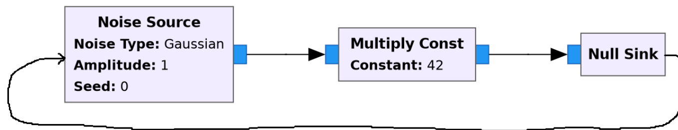
An alternative: closed circuits of “packets”

- Samples are sent in “packets” rather than as a continuous stream
- The flowgraph is divided into closed circuits, in which a fixed number of “packets” always exist
- “Packets” are recycled by sending them from a sink back to a source



An alternative: closed circuits of “packets”

- Samples are sent in “packets” rather than as a continuous stream
- The flowgraph is divided into closed circuits, in which a fixed number of “packets” always exist
- “Packets” are recycled by sending them from a sink back to a source

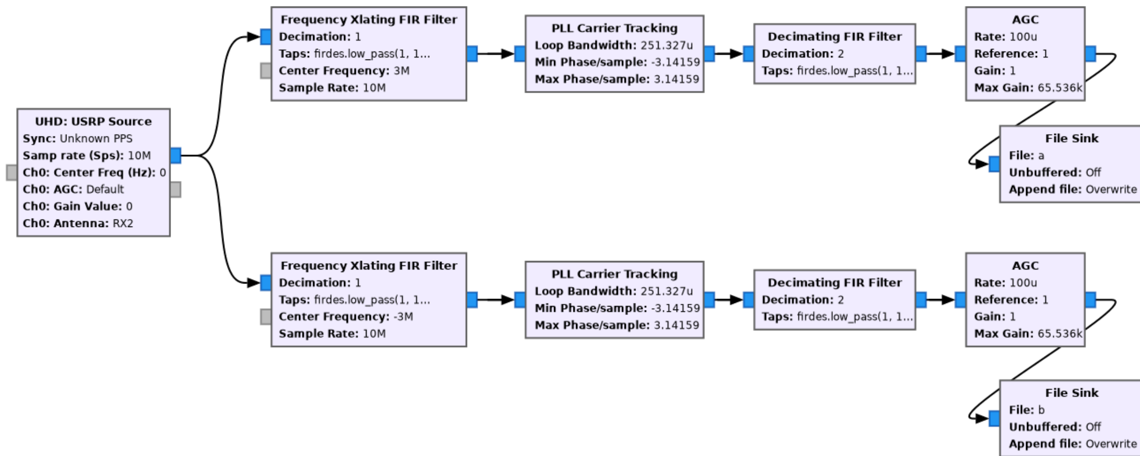


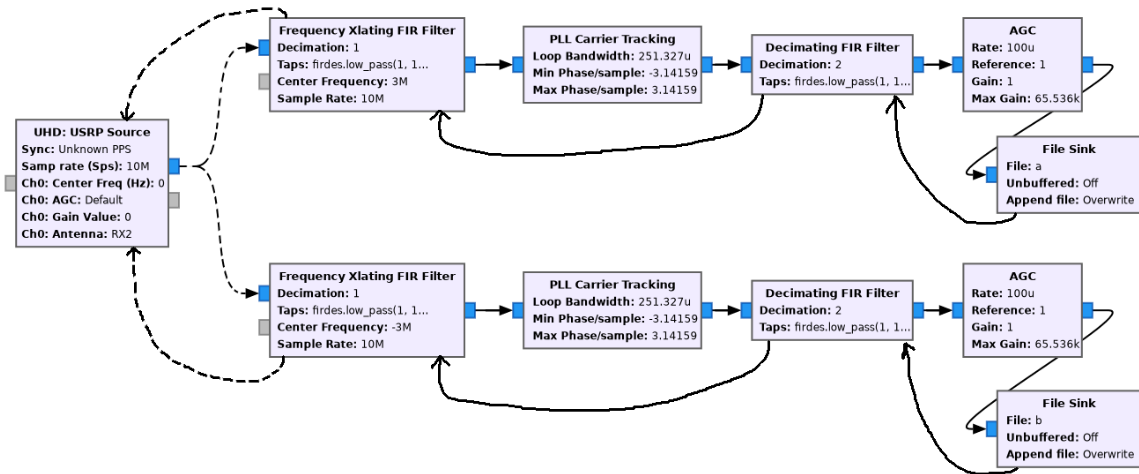
- Many blocks can work in-place on a “packet”
- Latency is determined by the number of “packets” in a circuit
- “Packets” can mark natural sections in the data (aligned to RF frame sections, etc.). Potentially less need for tags.
- More similar to a hand-crafted implementation where functions are called in sequence on the same buffer

The idea of a quantum

- Let's call this "packet" a **quantum**, because it is cool and because *packet* and *frame* are heavily overloaded
- A quantum contains:
    - A buffer with flexible *margins*. The margins allow inserting/adding a prefix/suffix in-place. Example use cases: CRC, synchronization word, cyclic prefix.
    - Tags, referred to sample indices within the packet
    - Perhaps other metadata that the user might need?

A more complex example flowgraph




UHD: USRP Source
Sync: Unknown PPS
Samp rate (Sps): 10M
Ch0: Center Freq (Hz): 0
Ch0: AGC: Default
Ch0: Gain Value: 0
Ch0: Antenna: RX2
Frequency Xlating FIR Filter
Decimation: 1
Taps: firdes.low_pass(1, 1...
Center Frequency: 3M
Sample Rate: 10M
PLL Carrier Tracking
Loop Bandwidth: 251.327u
Min Phase/sample: -3.14159
Max Phase/sample: 3.14159
Decimating FIR Filter
Decimation: 2
Taps: firdes.low_pass(1, 1...
AGC
Rate: 100u
Reference: 1
Gain: 1
Max Gain: 65.536k
File Sink
File: a
Unbuffered: Off
Append file: Overwrite
Frequency Xlating FIR Filter
Decimation: 1
Taps: firdes.low_pass(1, 1...
Center Frequency: -3M
Sample Rate: 10M
PLL Carrier Tracking
Loop Bandwidth: 251.327u
Min Phase/sample: -3.14159
Max Phase/sample: 3.14159
Decimating FIR Filter
Decimation: 2
Taps: firdes.low_pass(1, 1...
AGC
Rate: 100u
Reference: 1
Gain: 1
Max Gain: 65.536k
File Sink
File: b
Unbuffered: Off
Append file: Overwrite

qsdr

- `https://github.com/daniestevez/qsdr`
- An implementation of these ideas in Rust using `async`
- Some benchmarks comparing with GNU Radio 3.10, GNU Radio 4.0 and FutureSDR
- Still an experimental work-in-progress

Schedulers in qsdr

- GNU Radio 4.0 and FutureSDR support custom schedulers, but they are quite specific and not easy to write
- qsdr schedulers are based on Rust streams, so any code that can run streams can be a qsdr scheduler

- A stream is an object that can produce a sequence of values asynchronously

```rust
let mut stream = stream::iter(1..=3);

assert_eq!(stream.next().await, Some(1));
assert_eq!(stream.next().await, Some(2));
assert_eq!(stream.next().await, Some(3));
assert_eq!(stream.next().await, None);
```

- A stream is an object that can produce a sequence of values asynchronously

```
let mut stream = stream::iter(1..=3);

assert_eq!(stream.next().await, Some(1));
assert_eq!(stream.next().await, Some(2));
assert_eq!(stream.next().await, Some(3));
assert_eq!(stream.next().await, None);
```

- A qsdr block has a work function which is `async` (it awaits to get its inputs), processes one quantum per input/output, and returns `Run`, `Done` or an error
- A qsdr block is converted to a stream. Each `next()` call on this stream calls the work function once, and returns either an error or nothing.

- A stream is an object that can produce a sequence of values asynchronously

```
let mut stream = stream::iter(1..=3);

assert_eq!(stream.next().await, Some(1));
assert_eq!(stream.next().await, Some(2));
assert_eq!(stream.next().await, Some(3));
assert_eq!(stream.next().await, None);
```

- A qsdr block has a work function which is `async` (it awaits to get its inputs), processes one quantum per input/output, and returns `Run`, `Done` or an error
- A qsdr block is converted to a stream. Each `next()` call on this stream calls the work function once, and returns either an error or nothing.
- A qsdr scheduler is just some code that performs calls to multiple streams on one or several threads until all of them are done or there is an error

- Stream combinators are useful to build schedulers. For instance, `sequence2()` takes 2 streams (which produce items of type `Result<(), E>`) and produces a stream (with the same item type) that calls each of the 2 streams in sequence
- The helper function `run()` takes a stream and produces a future (an async result) that calls the stream until it is done or there is an error. This is usually the top-level element of the scheduler on each thread

```rust
type B = CacheAlignedBuffer<f32>;
let buffer_size = 4096;
let num_buffers = 4;

let head_elements = 100_000_000;

let buffers = std::iter::repeat_with(|| Quantum::new(B::new(buffer_size))).take(num_buffers);

let mut fg = Flowgraph::new();

let source = fg.add_block(NullSource::<Quantum<B>>::new());
let head = fg.add_block(Head::<Quantum<B>, Spsc, SpscRef>::new(head_elements));
let sink = fg.add_block(NullSink::<Quantum<B>>::new());

let mut circ = fg.new_circuit(buffers);
fg.connect(&mut circ, source.output(), head.input())?;
fg.connect_with_return(&mut circ, head.output(), sink.input(), source.input())?;

let mut fg = fg.validate()?;

let source = fg.extract_block(source)?;
let head = fg.extract_block(head)?;
let sink = fg.extract_block(sink)?;

println!("running flowgraph...");
block_on(run(sequence3(
    source.into_stream(),
    head.into_stream(),
    sink.into_stream(),
)))?;
println!("flowgraph finished");
```

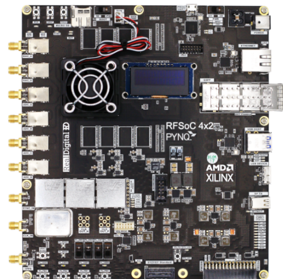
```rust
#[derive(Block, Debug)]
#[qsdr_crate = "crate"]
#[work(WorkInPlace)]
pub struct Head<T, Cin = Spsc, Cout = Spsc>
where
    Cin: Channel,
    Cin::Receiver<T>: Receiver<T>,
    Cout: Channel,
{
    #[port]
    input: PortIn<T, Cin>,
    #[port]
    output: PortOut<T, Cout>,
    remaining: u64,
}

impl<T, Cin, Cout> Head<T, Cin, Cout>
where
    Cin: Channel,
    Cin::Receiver<T>: Receiver<T>,
    Cout: Channel,
{
    pub fn new(count: u64) -> Self {
        Self {
            input: Default::default(),
            output: Default::default(),
            remaining: count,
        }
    }
}
```

```rust
impl<T, Cin, Cout> WorkInPlace<T> for Head<T, Cin, Cout>
where
    Cin: Channel,
    Cin::Receiver<T>: Receiver<T>,
    Cout: Channel,
{
    async fn work_in_place(&mut self, _: &mut T) -> Result<WorkStatus> {
        assert!(self.remaining > 0);
        self.remaining -= 1;
        if self.remaining == 0 {
            Ok(DoneWithOutput)
        } else {
            Ok(Run)
        }
    }
}
```

Benchmarking methodology

- Choose a family of simple flowgraphs
- Write by hand an implementation that performs as best as possible
- Write implementations in qsdr, GNU Radio 3.10, GNU Radio 4.0 and FutureSDR
- Measure the rate (samples/second) at which each implementation can run

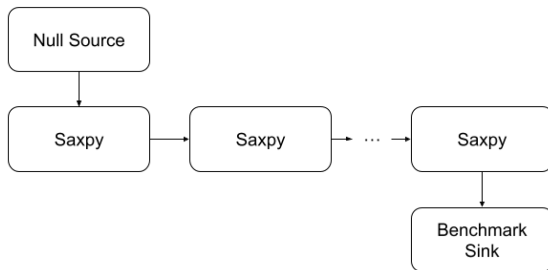
- Quad-core Cortex-A53 in AMD (Xilinx) MPSoC (1.33 GHz clock)
- Found in many AMD FPGA SoC platforms, in which a high-performance SDR runtime could be an alternative to an FPGA implementation for many signal processing problems



- Using a Kria KV260 board for development ($249 MSRP)

Benchmarking flowgraph

- Saxpy kernel: $y[n] = ax[n] + b$ (note: usual saxpy has a vector in place of b)



- Null Source does not do anything. Not even `memset()` the output to zero.
- Benchmark Sink counts samples and measures the sample rate

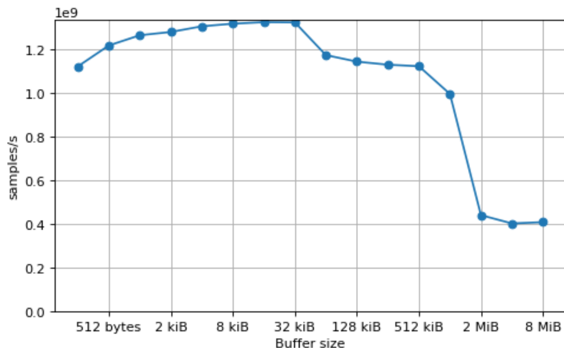
Saxpy kernel implementation

- Hand-written assembly using NEON SIMD
- Throughput of 1 float / clock cycle (2 FLOPs / clock cycle), which is the maximum according to the Cortex-A53 hardware limitations:
  - 2 NEON units each capable of operating on a 64-bit vector
  - 64-bit load path, 128-bit store path
  - Load/store to SIMD register uses corresponding NEON unit
  - One AGU for load/store
- For comparison, an optimal `memcpy()` is 1.33 floats / clock cycle (5.33 bytes / clock cycle)
- 2x as fast as the typical NEON code generated by gcc, clang and rustc, which is a 0.5 floats / clock cycle naïve implementation
- Heavily uses tricks related to the partial dual-issue capability of the Cortex-A53

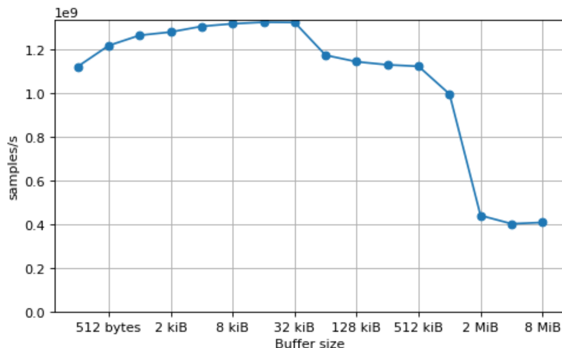
```
14068:  4c402944  ld1   {v4.4s-v7.4s}, [x10]
1406c:  4f919084  fmul  v4.4s, v4.4s, v17.s[0]
14070:  f9804140  prfm  pldl1keep, [x10, #128]
14074:  4f9190a5  fmul  v5.4s, v5.4s, v17.s[0]
14078:  f940254b  ldr   x11, [x10, #72]
1407c:  4f9190c6  fmul  v6.4s, v6.4s, v17.s[0]
14080:  f9402d4c  ldr   x12, [x10, #88]
14084:  4f9190e7  fmul  v7.4s, v7.4s, v17.s[0]
14088:  f940354d  ldr   x13, [x10, #104]
1408c:  4e30d484  fadd  v4.4s, v4.4s, v16.4s
14090:  f9403d4e  ldr   x14, [x10, #120]
14094:  4e30d4a5  fadd  v5.4s, v5.4s, v16.4s
14098:  f940394f  ldr   x15, [x10, #112]
1409c:  4e30d4c6  fadd  v6.4s, v6.4s, v16.4s
140a0:  f9804100  prfm  pldl1keep, [x8, #128]
140a4:  fd402140  ldr   d0, [x10, #64]
140a8:  4e181dc3  mov   v3.d[1], x14
140ac:  fd402941  ldr   d1, [x10, #80]
140b0:  4e181d60  mov   v0.d[1], x11
140b4:  fd403142  ldr   d2, [x10, #96]
140b8:  4e181d81  mov   v1.d[1], x12
140bc:  4e081de3  mov   v3.d[0], x15
140c0:  4e181da2  mov   v2.d[1], x13
140c4:  4e30d4e7  fadd  v7.4s, v7.4s, v16.4s
140c8:  4f919000  fmul  v0.4s, v0.4s, v17.s[0]
140cc:  f940454b  ldr   x11, [x10, #136]
140d0:  4f919021  fmul  v1.4s, v1.4s, v17.s[0]
140d4:  4f919042  fmul  v2.4s, v2.4s, v17.s[0]
140d8:  f9404d4c  ldr   x12, [x10, #152]
140dc:  4f919063  fmul  v3.4s, v3.4s, v17.s[0]
140e0:  f940554d  ldr   x13, [x10, #168]
140e4:  4e30d400  fadd  v0.4s, v0.4s, v16.4s
140e8:  f9405d4e  ldr   x14, [x10, #184]
140ec:  4e30d421  fadd  v1.4s, v1.4s, v16.4s
140f0:  f940594f  ldr   x15, [x10, #176]
140f4:  4c002944  st1   {v4.4s-v7.4s}, [x10]
140f8:  fd404945  ldr   d5, [x10, #144]
140fc:  4e181dc7  mov   v7.d[1], x14
14100:  fd405146  ldr   d6, [x10, #160]
14104:  4e181d85  mov   v5.d[1], x12
14108:  fc480d44  ldr   d4, [x10, #128]!
```

```
1410c:  4e181da6  mov   v6.d[1], x13
14110:  4e081de7  mov   v7.d[0], x15
14114:  4e181d64  mov   v4.d[1], x11
14118:  4e30d442  fadd  v2.4s, v2.4s, v16.4s
1411c:  f9806100  prfm  pldl1keep, [x8, #192]
14120:  4e30d463  fadd  v3.4s, v3.4s, v16.4s
14124:  f9808100  prfm  pldl1keep, [x8, #256]
14128:  4f919084  fmul  v4.4s, v4.4s, v17.s[0]
1412c:  f940450b  ldr   x11, [x8, #136]
14130:  4f9190a5  fmul  v5.4s, v5.4s, v17.s[0]
14134:  4f9190c6  fmul  v6.4s, v6.4s, v17.s[0]
14138:  f9404d0c  ldr   x12, [x8, #152]
1413c:  4f9190e7  fmul  v7.4s, v7.4s, v17.s[0]
14140:  f940550d  ldr   x13, [x8, #168]
14144:  4e30d484  fadd  v4.4s, v4.4s, v16.4s
14148:  f9405d0e  ldr   x14, [x8, #184]
1414c:  4e30d4a5  fadd  v5.4s, v5.4s, v16.4s
14150:  f940590f  ldr   x15, [x8, #176]
14154:  4c002900  st1   {v0.4s-v3.4s}, [x8]
14158:  fd404901  ldr   d1, [x8, #144]
1415c:  4e181dc3  mov   v3.d[1], x14
14160:  fd405102  ldr   d2, [x8, #160]
14164:  4e181d81  mov   v1.d[1], x12
14168:  fc480d00  ldr   d0, [x8, #128]!
1416c:  4e181da2  mov   v2.d[1], x13
14170:  4e081de3  mov   v3.d[0], x15
14174:  4e181d60  mov   v0.d[1], x11
14178:  4e30d4c6  fadd  v6.4s. v6.4s. v16.4s
1417c:  eb09015f  cmp   x10, x9
14180:  4e30d4e7  fadd  v7.4s, v7.4s, v16.4s
14184:  54fffa21  b.ne  140c8
14188:  4f919000  fmul  v0.4s, v0.4s, v17.s[0]
1418c:  4f919021  fmul  v1.4s, v1.4s, v17.s[0]
14190:  4f919042  fmul  v2.4s, v2.4s, v17.s[0]
14194:  4f919063  fmul  v3.4s, v3.4s, v17.s[0]
14198:  4c002944  st1   {v4.4s-v7.4s}, [x10]
1419c:  4e30d400  fadd  v0.4s, v0.4s, v16.4s
141a0:  4e30d421  fadd  v1.4s, v1.4s, v16.4s
141a4:  4e30d442  fadd  v2.4s, v2.4s, v16.4s
141a8:  4e30d463  fadd  v3.4s, v3.4s, v16.4s
141ac:  4c002900  st1   {v0.4s-v3.4s}, [x8]
```

Saxpy kernel benchmark depending on buffer size



Saxpy kernel benchmark depending on buffer size



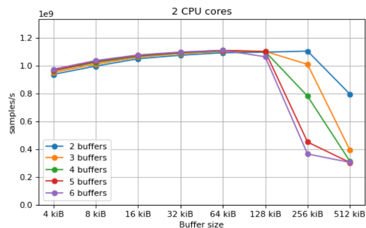
- 32 KiB L1 cache (per core), 1 MiB L2 cache (shared by all cores)

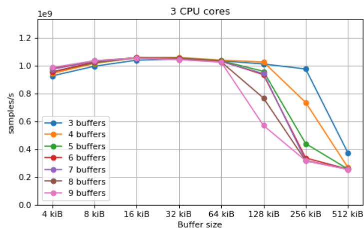
Some comments about SDR runtime performance

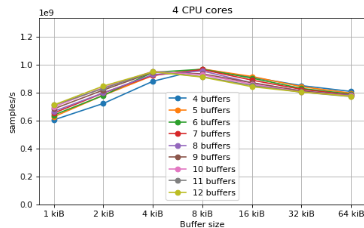
- We want most of the CPU time to be spent on running our `work()` functions, rather than something else
- SDR runtime overhead depends on number of `work()` calls per second
- Using large buffers for each `work()` call is not an option. Ideally we want to stay in L1 cache (and definitely not go to DDR).
- Fast simple kernels which can process an L1 cache worth of data quickly are the worst case, since SDR runtime overhead can be significant
- Some numbers for Saxpy kernel: 16 KiB buffer $\rightarrow$ 4096 floats. $\sim$1 float/cycle at 1.33 GHz $\rightarrow$ 3 μs per `work()` call.

Saxpy kernel benchmark in multiple CPU cores

- Rust channels used to send buffers between threads. Custom high-performance channels implemented in qsdr.
- One thread pinned to each CPU core
- Fixed number of buffers passed around in a circuit formed by the threads

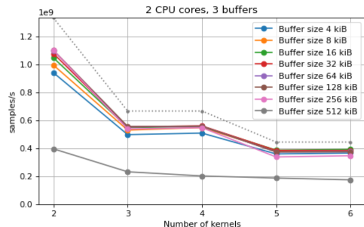


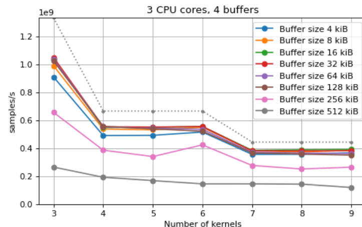


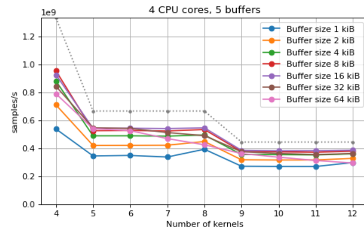


Saxpy kernel benchmark of multiple kernels

- *M* kernels and *N* CPU cores. Kernels statically pinned to CPU cores. Worst case, $\text{ceil}(M/N)$ kernels sharing a core.

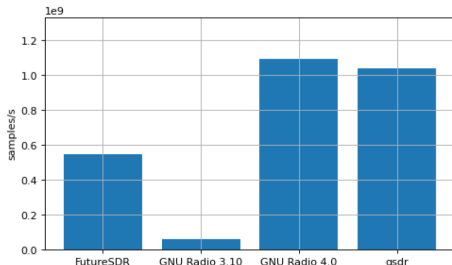




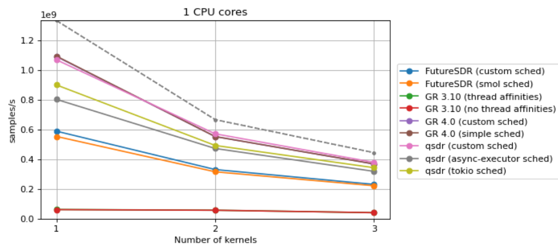


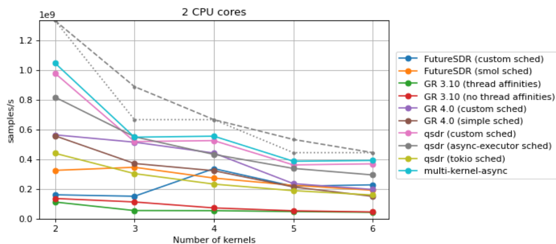
SDR runtime benchmark with a single Saxpy kernel and single core

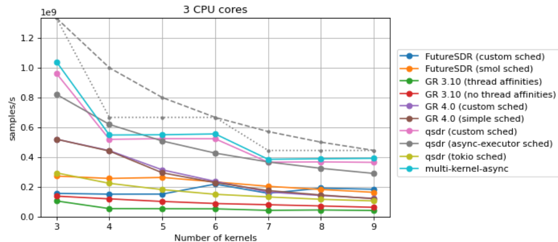
- Flowgraph: null source $\rightarrow$ saxpy $\rightarrow$ benchmark sink
- Uses simplest scheduler to run all the blocks in the same CPU

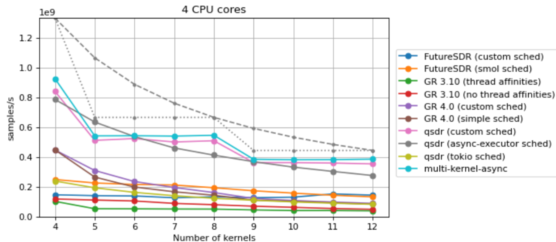


SDR runtime benchmark of multiple Saxpy kernels









Conclusions

- There is a lot of room for improvement in SDR runtime performance
- What is the future of qsdr?
  - Hard to say at this point. Might develop further or remain as an experiment
  - Currently more intended as a source of ideas and to compare with other SDR runtimes
  - If it develops further, the goal would be a lower-level runtime than GNU Radio. A middle ground between having many things done/chosen for you versus having to write everything from scratch.