Holoscan and USRP: A GPU and SDR Case Study

Martin Braun

Ettus Research / NI / Emerson

Sage Trudeau

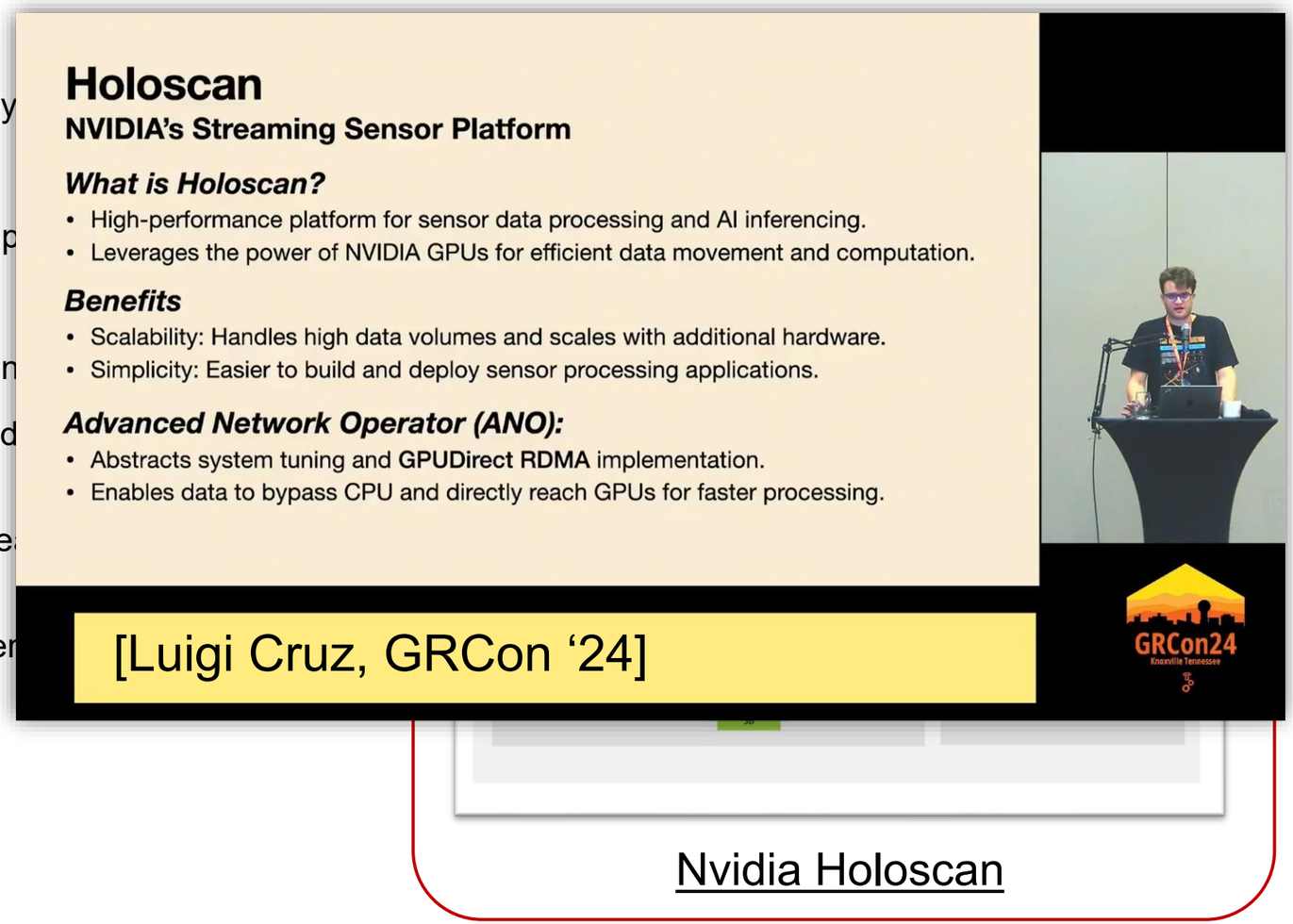
UT Austin

Holoscan and DAQIRI

What’s a Daqiri? Is it already that time of day?

What are Holoscan and DAQIRI?

- Holoscan: A data movement and GPU processing framework by
  - Open Source, Permissively Licensed
  - Composable architecture, block-based design (“one block p
  - All right, let’s be real, this is GNU Radio for GPUs
  - Ideal for Ethernet-based I/O, good for streaming application
  - Developers focus on building IP blocks (GPU algorithms), d
    memory management is handled by the framework
  - Can be used with Ethernet-capable devices, USRP is a gre
    interoperability
  - Has a Python and a C++ API, those are not (yet?) interoper
- Daqiri: The I/O library that handles data movement
  - “Data Acquisition for Integrated Real-time Instruments”
  - Was originally part of Holoscan, and they connect well



Nvidia Holoscan

GNU Radio <-> Holoscan Comparison

GNU Radio

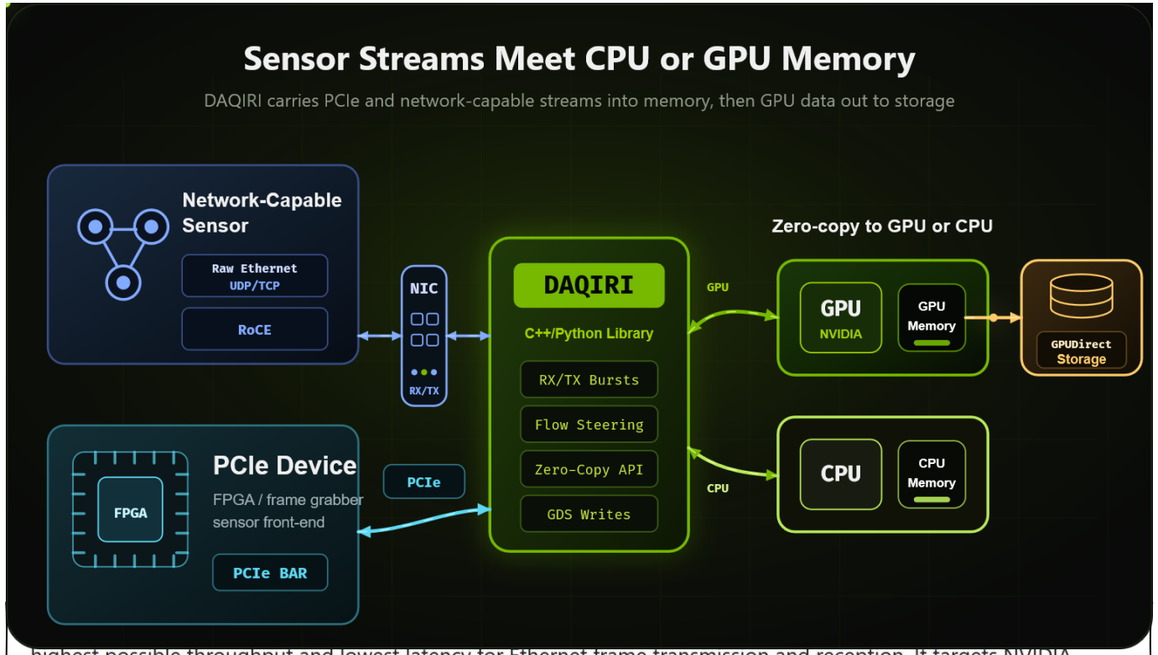
- Scheduler stems from the early 2000s, but very stable API
- CPU focused, very portable
- Focus on radio/DSP
- Switch between C++ API / Python API / GRC, combine components from all language domains
- Very mature and comprehensive block library, both in-tree and out-of-tree
- Between CGRAN and github: Easy to find out-of-tree modules

Holoscan

- Modern architecture, but APIs change frequently
- Made for use with Nvidia GPUs
- Comes from medical imaging (?), applications very varied
- No GUI whatsoever, even visualizations are highly limited, C++/Python not interoperable
- Extension community not well established, Nvidia not focused on building out open-source block library
- Holohub is the main extension repository, where all users upstream their extensions/applications/examples

Daqiri: Abstracting the I/O

- Assumption: You are using lots of Nvidia-compatible hardware (GPU, ConnectX NIC, ideally DGX/IGX/RTX machines)
- Daqiri abstracts different transport modes (e.g., RDMA, raw UDP + DPDK, …) to easily enable GPUDirect support for your external sensor (e.g., your SDR)
- If you get it installed on your system, and the hardware allows it, then you can easily stream data rates of 2000 Msps (8 Gbyte/s) or more directly into your GPU memory space
- Runtime configuration options are provided through a YAML file



highest possible throughput and lowest latency for Ethernet frame transmission and reception. It targets NVIDIA ConnectX-6 Dx and later NICs and supports GPU direct memory access (GPUDirect) for zero-copy data paths between the NIC and GPU.

📖	**Docs & Website:** nvidia.github.io/daqiri
⚡	**Peak performance** requires an NVIDIA SmartNIC (ConnectX-6 Dx or later) and a GPUDirect-capable NVIDIA GPU
🖥️	**Supported hardware:** NVIDIA DGX Spark, NVIDIA IGX, and NVIDIA RTX Pro Servers
🔌	**Works with any NIC and NVIDIA GPU** via DAQIRI's built-in Linux Sockets engine
🚀	**Getting Started:** nvidia.github.io/daqiri/getting-started

Examples: config.yml and main app

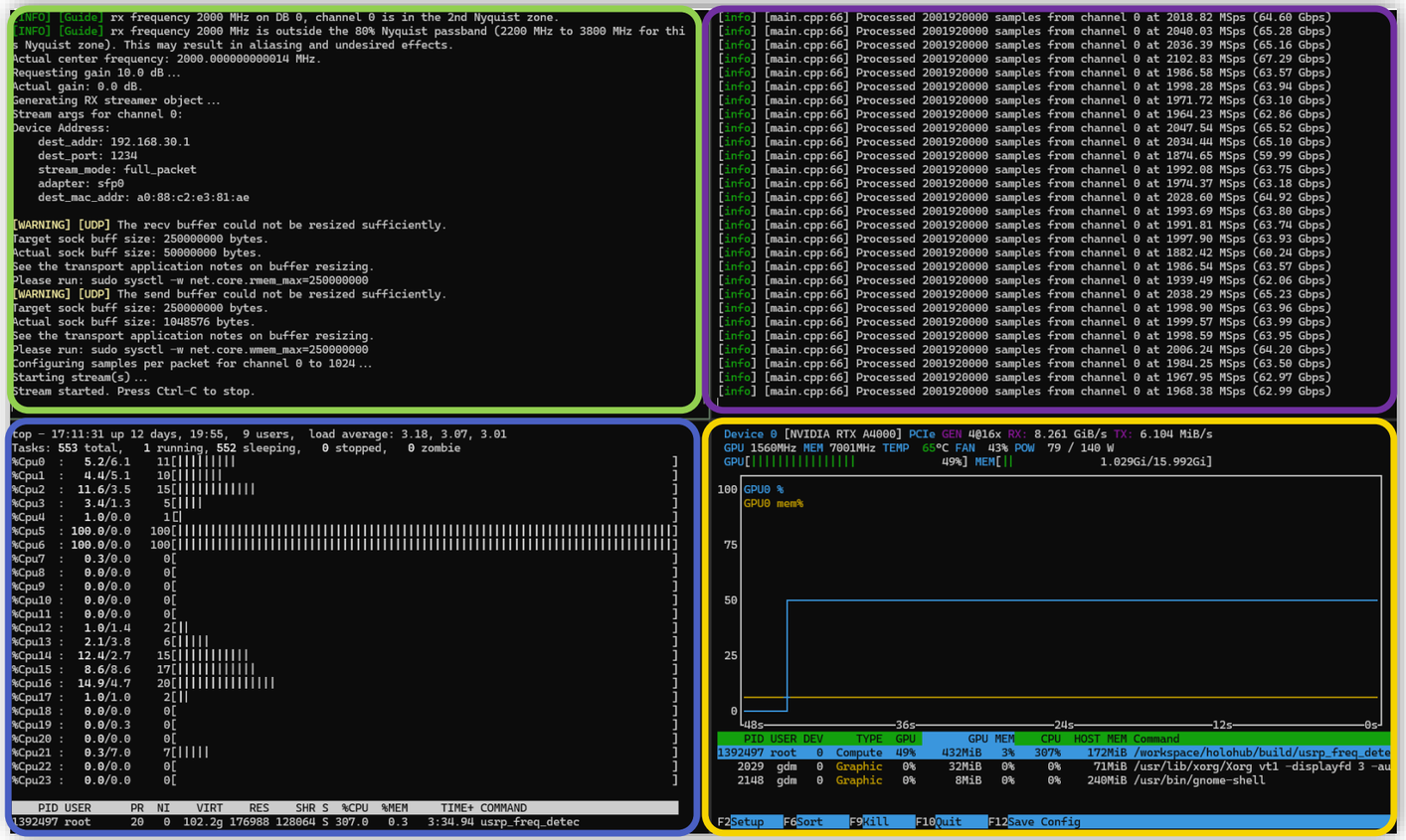
```yaml
# SPDX-FileCopyrightText: 2026 National Instruments Corporation
#
# SPDX-License-Identifier: Apache-2.0
---
num_runs: -1  # Number of iterations the application will run, -1 is infinite

scheduler:
  worker_thread_number: 4
  stop_on_deadlock: true
  stop_on_deadlock_timeout: 500
  enable_worker_postcheck_fastpath: true
  enable_queue_stealing: true

usrp_rx:
  # Need to be set to true to enable data transfer from USRP.
  # Make sure that the "<device_addr>", "<dest_addr>", "<dest_mac_addr>" below are set correctly to your target setu
  enabled: false
  # Set your real device args before enabling, e.g.:
  # args: "addr=10.88.136.10,master_clock_rate=500e6"
  args: "addr=<1.2.3.4>,master_clock_rate=500e6"
  rate: 500e6  # Hz (500 MHz); recommended to match spectrum_viz.ref_bandwidth_hz
  freq: 1000e6 # Hz (1000 MHz); recommended to match spectrum_viz.ref_center_hz
  gain: 0.0
  channels: [0, 1]
  # Set the DPDK host IP before enabling.
  dest_addr: <5.6.7.8> # Destination IP address
  dest_ports: [1234, 1235]
  adapter: "sfp0"
  dest_mac_addr: <00:00:00:00:00:00> # Destination MAC address
  keep_hdr: true
  spp: 1024
  mtu: 8064
  start_delay_seconds: 0.05

daqiri:
  cfg:
    version: 1
    stream_type: "raw"
    master_core: 9  # Master CPU core
    debug: false
    log_level: "info"

    memory_regions:
      # Memory pool for frame headers
      - name: "Headers_RX_CPU"
        kind: "huge"
        affinity: 0
        access:
          - local
        num_bufs: 12500
        # Size of the Ethernet header + IPv4 header + UDP header
        buf_size: 42
      # Memory pool for channel 1 CHDR packet headers
      - name: "CH1_CHDR_Headers_RX_CPU"
        kind: "huge"
        affinity: 0
        access:
          - local
        num_bufs: 12500
        # Size of the CHDR header (depends on CHDR_W size)
```

```cpp
#include "usrp_rx/usrp_rx.hpp"
#include "uhd_chdr_rx/uhd_chdr_rx.hpp"
#include "log/log.hpp"
#include "spectrum_overlay/spectrum_overlay.hpp"
#include "spectrum_visualizer/spectrum_visualizer.hpp"
#include "spectrum_magnitude/spectrum_magnitude.hpp"
#include <fft.hpp>
#include <holoscan/operators/holoviz/holoviz.hpp>
#include <memory>

class UsrpSpectrumViewerApp : public holoscan::Application {
 public:
    void compose() override {
        using namespace holoscan;

        // Shared state couples the Holoviz overlay controls with the spectrum
        // visualizer's pan/zoom and peak reporting logic.
        auto view_state = std::make_shared<ops::SpectrumViewState>();
        auto overlay_config = apps::SpectrumOverlayConfig{
            .db_min = from_config("spectrum_viz.db_min").as<float>(),
            .db_max = from_config("spectrum_viz.db_max").as<float>(),
            .initial_center_freq_mhz =
                from_config("spectrum_viz.ref_center_hz").as<float>() * 1.0e-6F,
            .initial_bandwidth_mhz =
                from_config("spectrum_viz.ref_bandwidth_hz").as<float>() * 1.0e-6F,
            .num_channels = from_config("spectrum_viz.num_channels").as<int>()};

        auto usrp_rx = make_operator<ops::UsrpRxOp>(
            "usrp_rx",
            from_config("usrp_rx"));

        auto uhd_chdr_rx = make_operator<ops::UhdChdrRxOp>(
            "uhd_chdr_rx",
            from_config("uhd_chdr_rx"));

        auto fft = make_operator<ops::FFT>(
            "fft",
            from_config("fft"));

        auto log = make_operator<ops::LogOp>(
            "log",
            from_config("log"),
            make_condition<CountCondition>(from_config("num_runs").as<int64_t>()));

        auto spectrum_magnitude = make_operator<ops::SpectrumMagnitudeOp>(
            "spectrum_magnitude",
            from_config("spectrum_magnitude"));

        // Managed stream pool lets the framework event-sync with Holoviz (no host block).
        auto spectrum_viz = make_operator<ops::SpectrumVisualizerOp>(
            "spectrum_viz",
            from_config("spectrum_viz"),
            make_resource<CudaStreamPool>("spectrum_viz_stream_pool", 0, 0, 0, 1, 5));
        // The visualizer reads live pan/zoom and peak-toggle state from the UI.
        spectrum_viz->set_view_state(view_state);
```

Holoscan-Based Applications

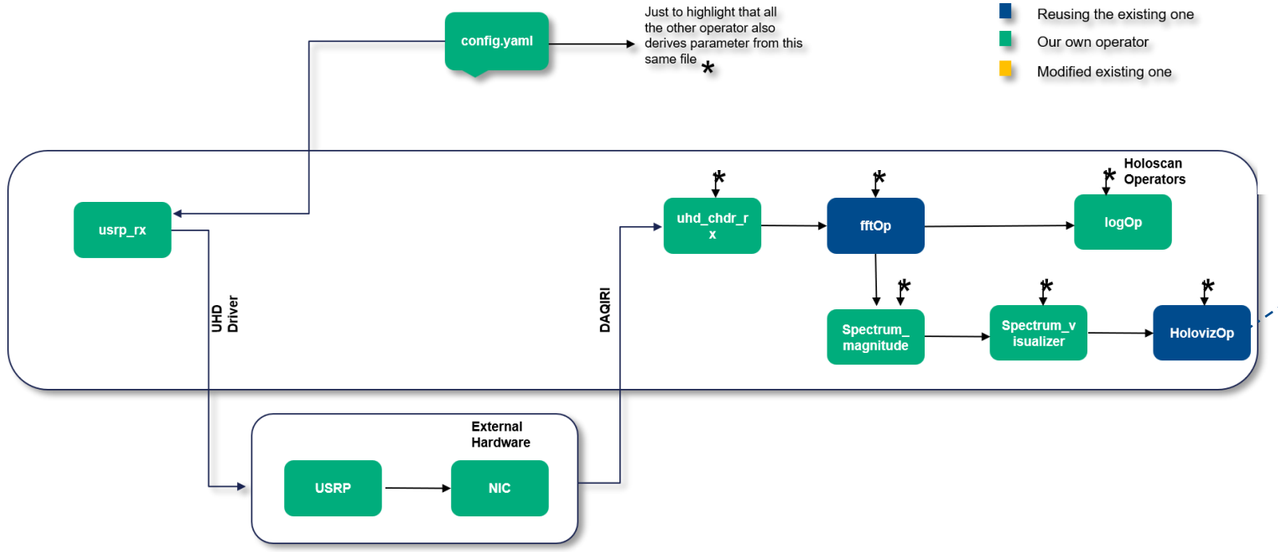
It all started with a PoC…

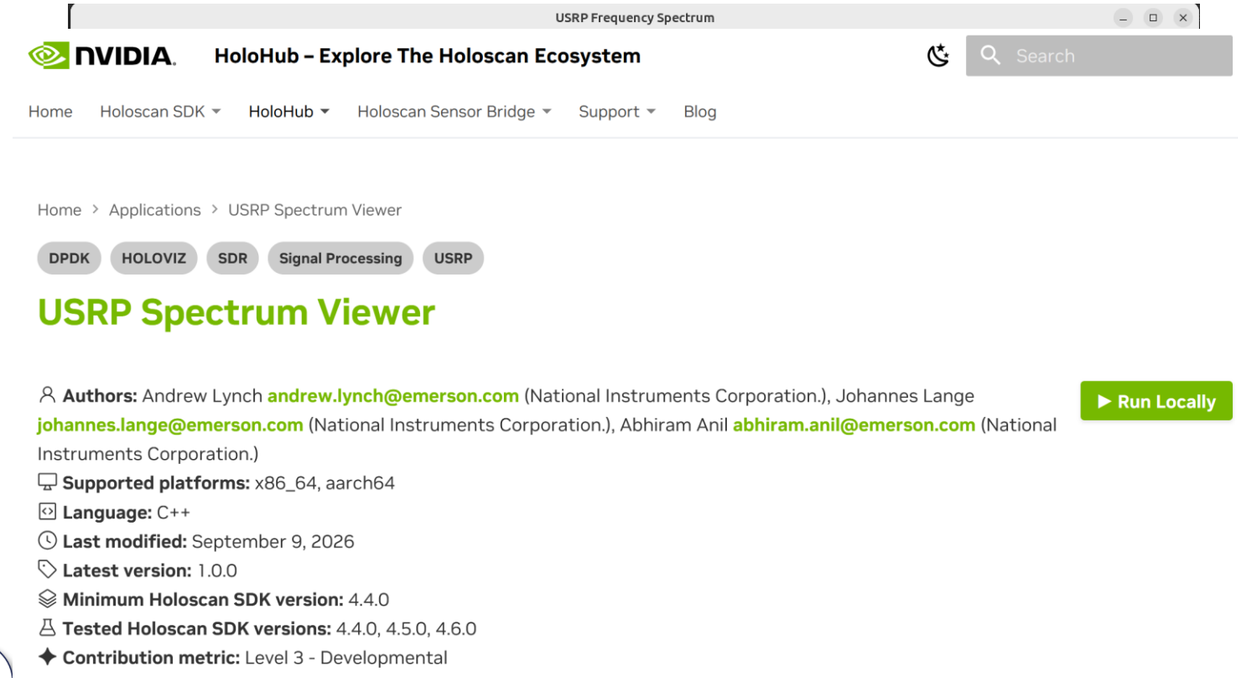
- We wanted to see if Holoscan gives us anything useful
- We used the raw UDP streaming feature of the USRP to produce a high-rate output that could be ingested by Holoscan
- Various platforms were tested
- As a stand-in for an SDR app, we simply ran a 20k FFT
- Full rate of X440 (2000 Msps) could be streamed without losing data
- Uses DPDK for data transfer, blocks CPU cores
- This could be useful!



USRP Spectrum Viewer

- Spectrum Viewer: Super-simple demo that enables SDR users to get started with Holoscan
- Uses Holoviz for visualization
- No sample ever leaves GPU memory!
- Available on Github/Holohub! (Thanks to our intern Abhiram!)





EMERSON

Limitations

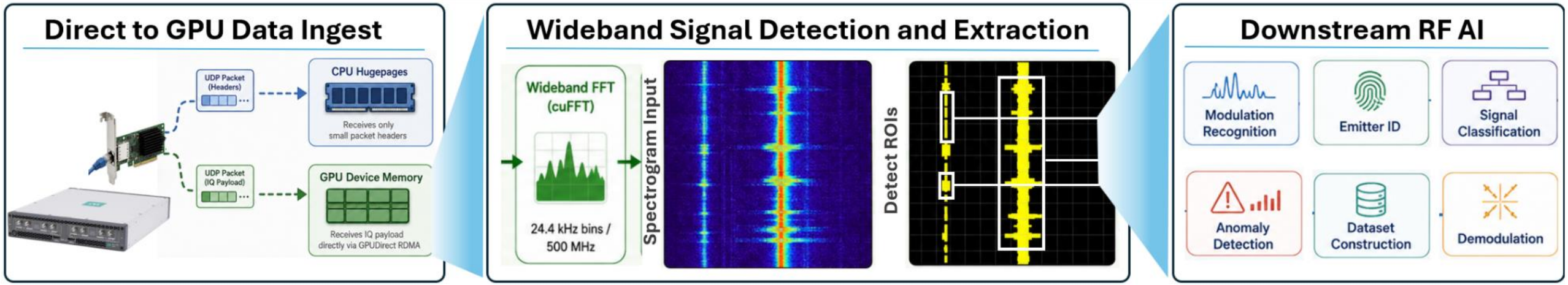
- Currently, lots of limitations:
  - No flow control support: You must make sure the GPU algorithms can handle all incoming data, or data is lost and bad things happen
  - No Tx support: Consequence of no flow control

TEXAS
The University of Texas at Austin

ARGUS

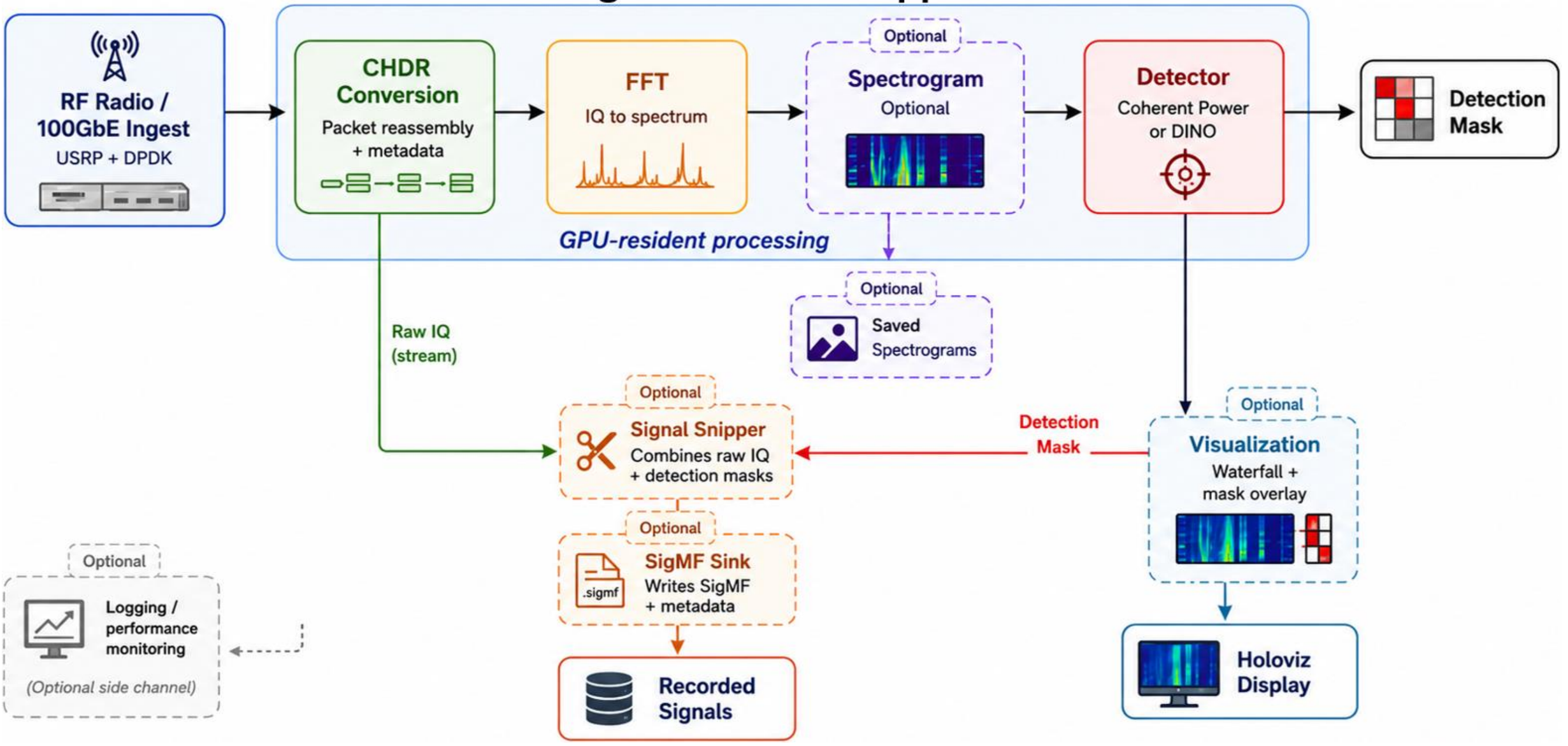
SAGE TRUDEAU, BRANDON NGUYEN, DIVYADHARSHINI MURUGANANDHAM, KAUSHIK CHOWDHURY

How do we deploy real time RF-AI models in the wild?



ARGUS addresses whether the end-to-end wideband system can keep up with data delivery to RF AI models at all

Holoscan Signal Detection Application Flow



Control Setup

USRP X410

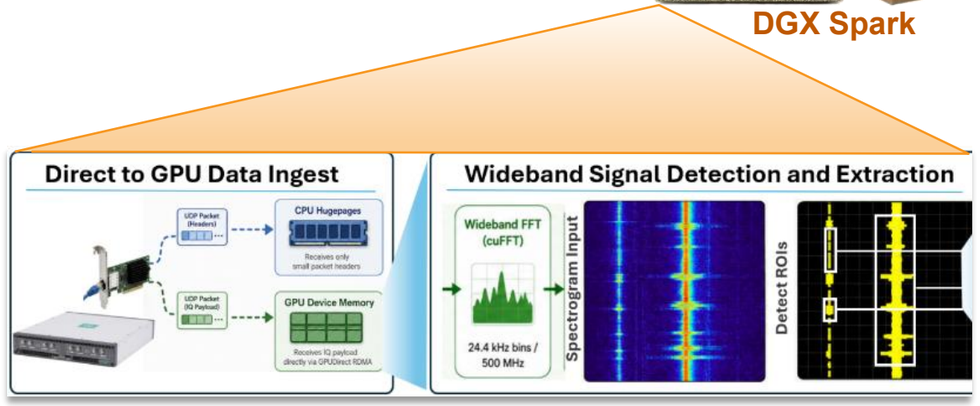
RF Coax

USRP X410

QSFP+ 100GE

DGX Spark

Waveform Library

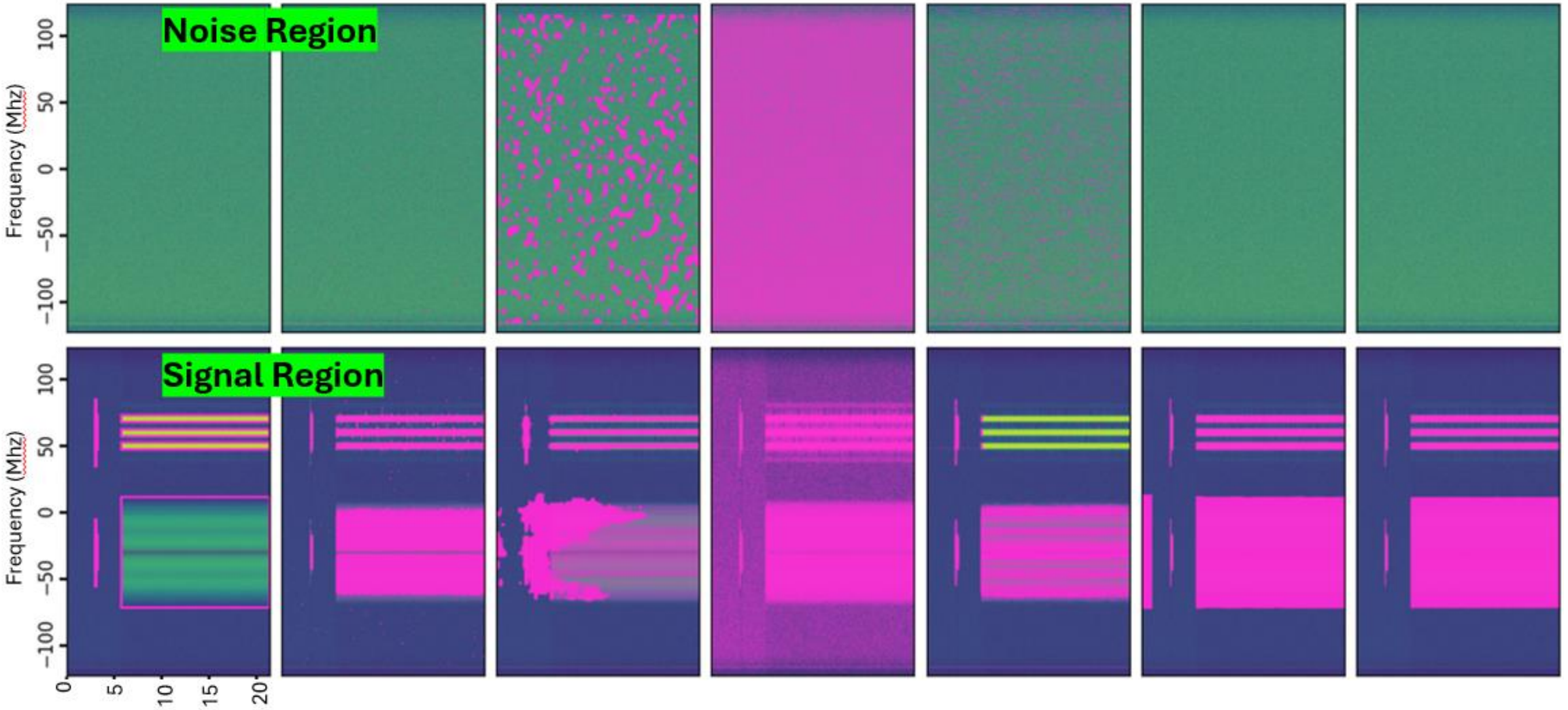


ARGUS Framework

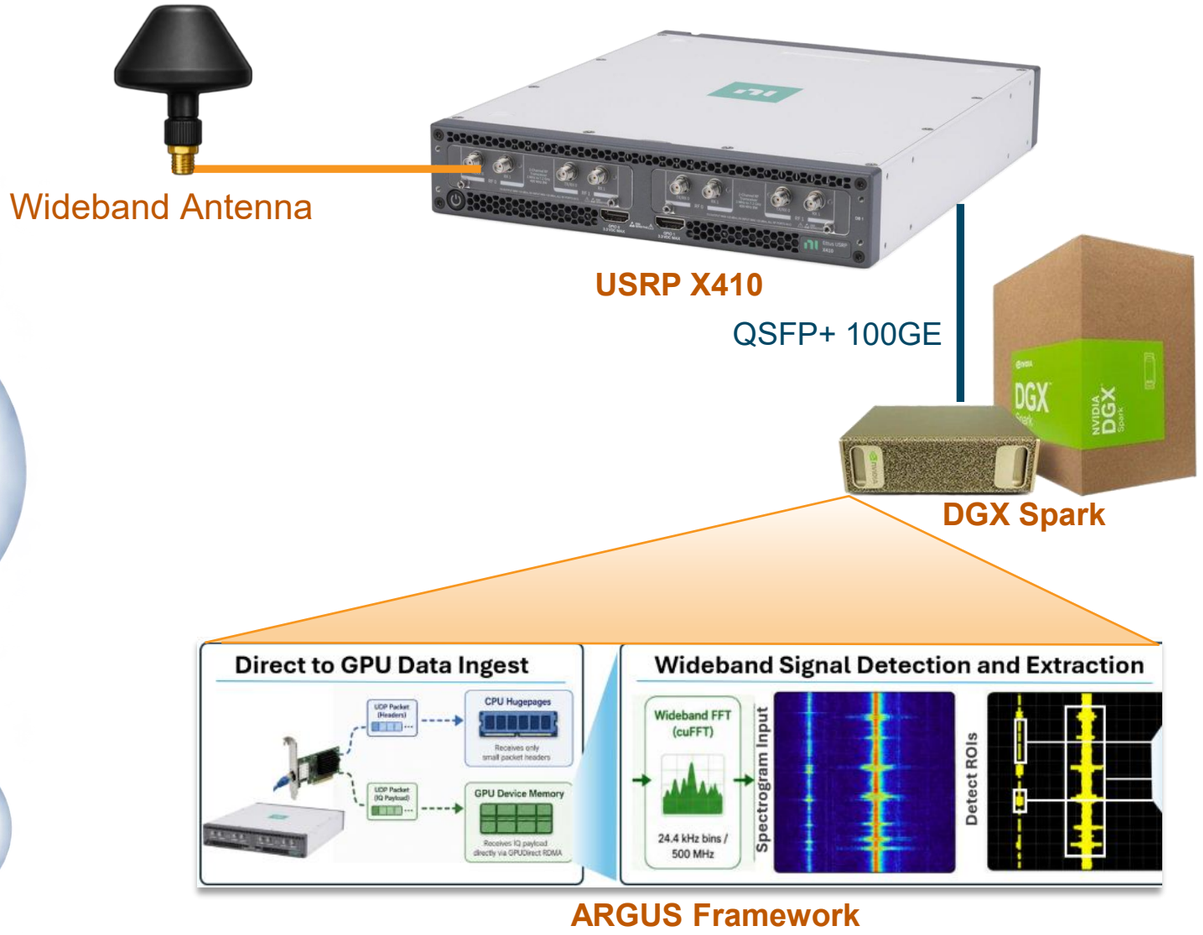
Table 2: Representative coverage of the controlled ARGUS evaluation library.

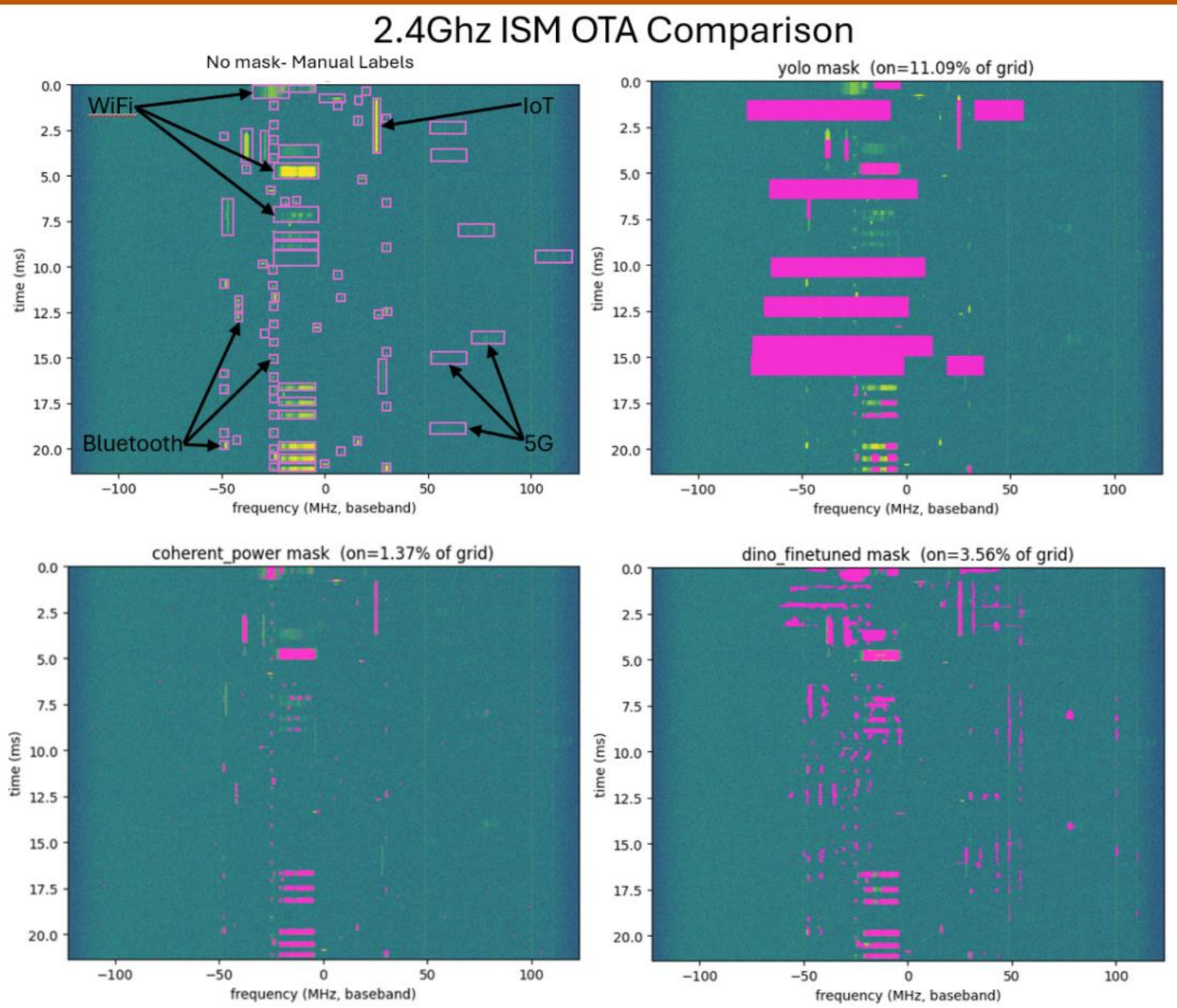
Family	Approx. occupied bandwidth	Detection challenge
BPSK/QPSK/16QAM	0.29–82.94 MHz	Single-carrier PSK/QAM over a broad bandwidth range
OFDM	4.50–98.28 MHz	Structured multicarrier emissions
5G NR downlink	3.96–98.28 MHz	Standardized wideband multicarrier traffic
802.11ax	20–160 MHz	Very wide Wi-Fi emissions
Bluetooth	1–2 MHz	Narrow frequency-hopping/GFSK activity
Broadband FM	0.13–61.47 MHz	Analog FM over highly variable bandwidth
Narrowband FM	0.01–0.03 MHz	Extremely thin spectral support

Qualitative Detector Comparison



Deployment Setup



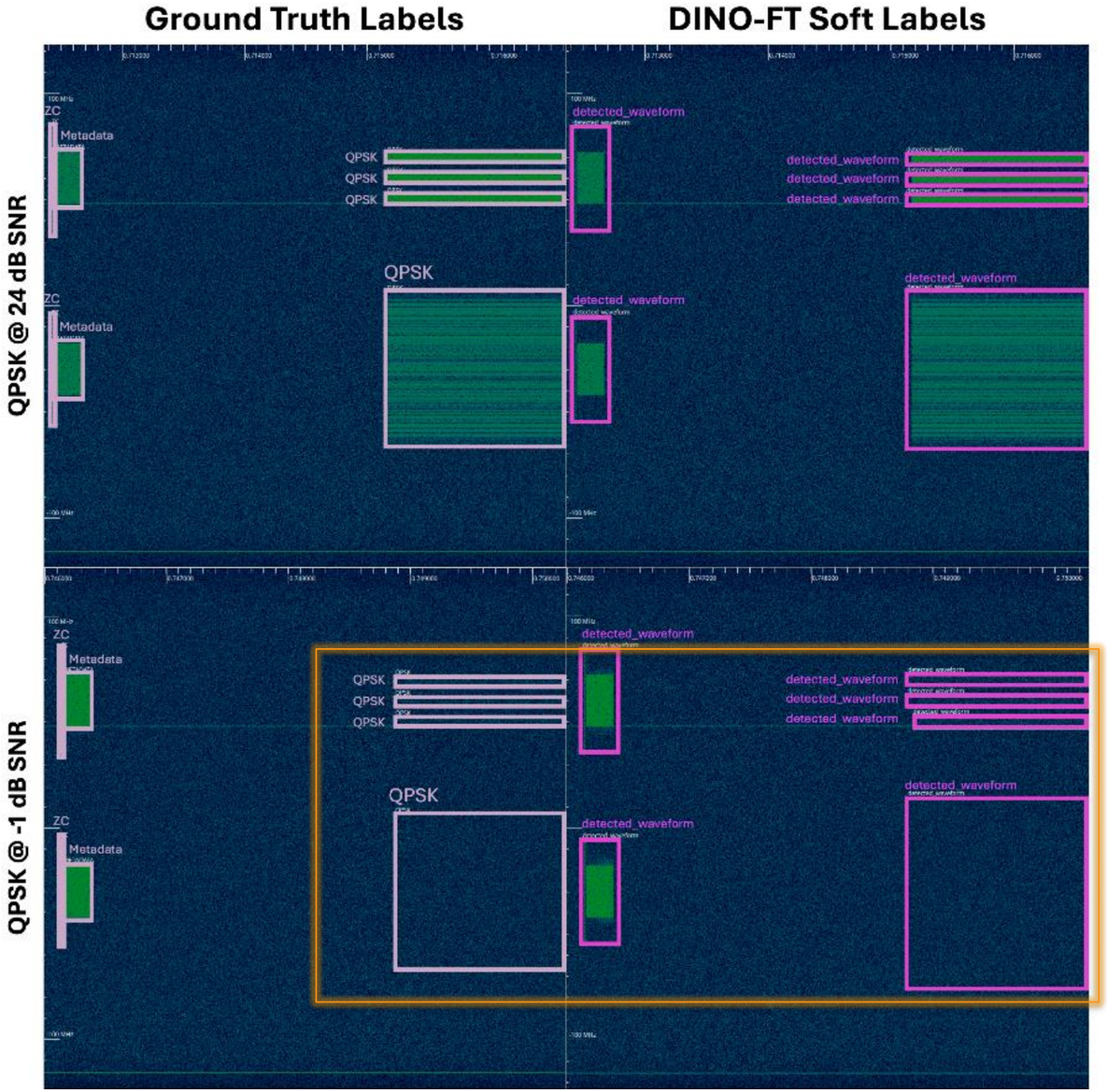


Out of Distribution Environment

- Fine-tuned DINO localizes several weaker structures near the visible noise floor
- Coherent power primarily captures strong emissions
- YOLO produces larger box-shaped regions and misses several narrowband structures.

Low SNR Conditions

- Can you see the QPSK signal in the orange region?
- DINO can!



DINO-FT shows impressive capability to localize signal region below the noise

Enabling RF AI for Ultra Wideband Real Time Applications

Holoscan enables GPU native real time signal processing and computer vision models to detect and extract regions of interest from dual 500 MSps streams, reducing RF AI workloads to only the regions that matter.

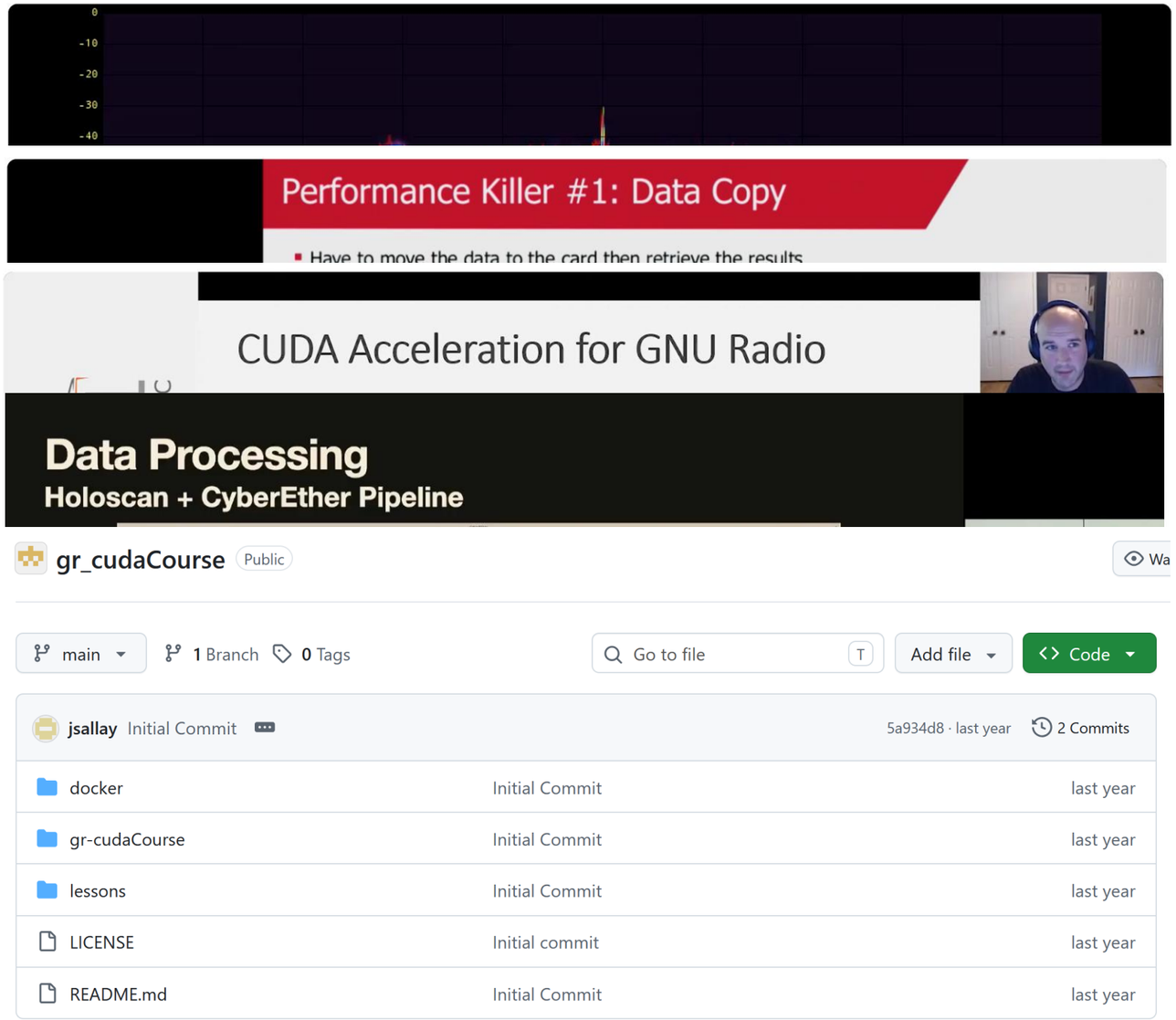
GNU Radio and GPUs

EMERSON

A brief history…

- GPUs and SDRs go way back, but the first “big bang” was probably fosphor (and gr-fosphor) by Sylvain (OpenCL-based)
- Mike P. (among others!) talked about accelerating blocks with GPUs as early as 2017
- DARPA + BlackLynx bring us custom buffers, and with that comes gr-cuda
- Adam T. and his gang have been coming to GRCon for a while now, showing off new tools for use with SDRs
- Luigi keeps building crazy cool apps that build on GPUs (among other things)
- JohnS and Beecubed provide tutorials for gr-cuda
- I probably forgot loads of people

- Kudos, shoutouts, respect, and all the appropriate credit to all these folks!



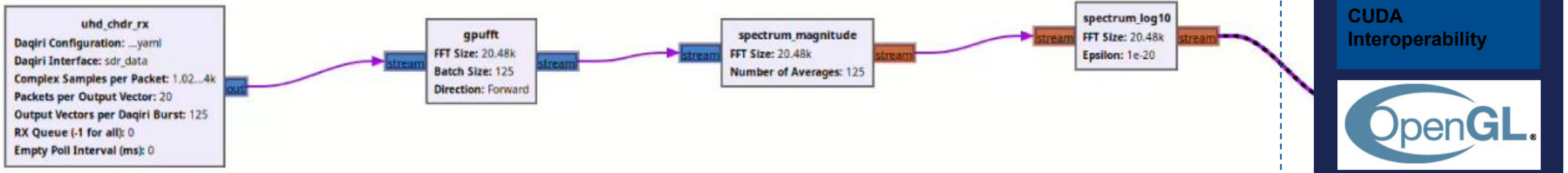
EMERSON

gr-cuda ❤ Daqiri

gr-cuda

https://github.com/gnuradio/gr-cuda

daqiri



GPU | CPU

CUDA Interoperability

OpenGL

Goal
- Move, process and visualize I/Q data (500 Msps). All on GPU!

Current State
- Integrated DAQIRI into GNU Radio ecosystem
- Created FFT, magnitude and log conversion blocks (similar to Holoscan)
- Able to achieve 90% full functional GPU pipeline, “only” visualization missing

What’s missing?
- Idea: visualization using CUDA interop (CUDA + openGL)
- Add remote streaming configuration as GNU radio block

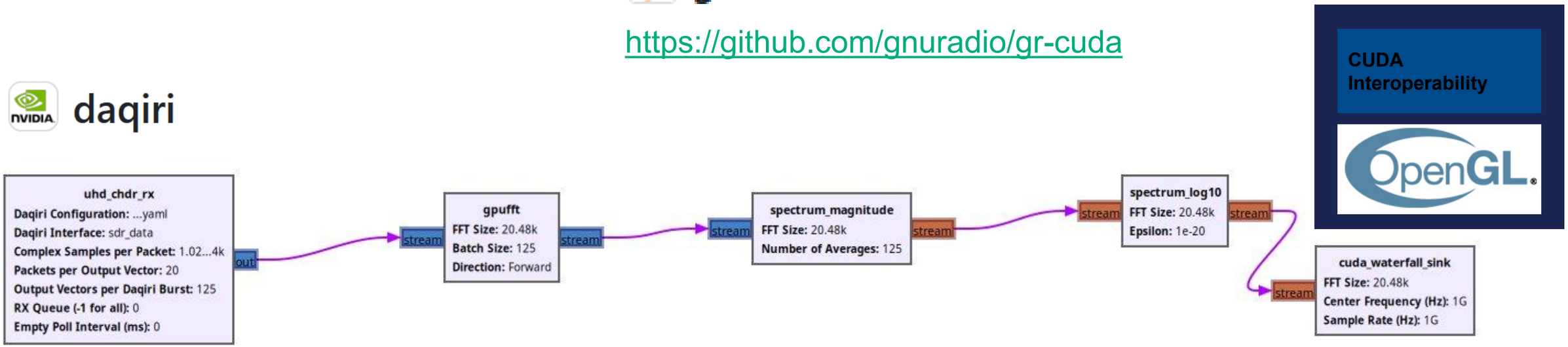
EMERSON

gr-cuda ❤ Daqiri

gr-cuda

https://github.com/gnuradio/gr-cuda

daqiri



Goal
- Move, process and visualize I/Q data (500 Msps). All on GPU!

Current State
- Integrated DAQIRI into GNU Radio ecosystem
- Created FFT, magnitude , log conversion and visualization blocks (similar to Holoscan)
- Able to achieve 100% full functional GPU pipeline with maximum I/Q data (1500Msps)

What's missing?
- Add remote streaming configuration as GNU radio block

EMERSON

That’s all for now

- Bottom line: There are great ways for SDR/GPU integration
- This is all pretty new
- gr-cuda blocks: We still need to figure out open-sourcing mechanics

- Please pester us with questions!