Aerial CUDA-Accelerated RAN

Table of contents

List of Figures

List of Tables

Aerial CUDA-Accelerated RAN brings together the Aerial software for 5G and AI frameworks and the NVIDIA accelerated computing platform, enabling TCO reduction and unlocking infrastructure monetization for telcos.

Aerial CUDA-Accelerated RAN has the following key features:

- Software-defined, scalable, modular, highly programmable and cloud-native, without any fixed function accelerators. Enables the ecosystem to flexibly adopt necessary modules for their commercial products.
- Full-stack acceleration of DU L1, DU L2+, CU, UPF and other network functions, enabling workload consolidation for maximum performance and spectral efficiency, leading to best-in-class system TCO.
- General purpose infrastructure, with multi-tenancy that can power both traditional workloads and cutting-edge AI applications for best-in-class RoA.

What's New in 24-1

Now Available in Release 24-1 for Aerial CUDA-Accelerated RAN

- **Aerial cuPHY**: CUDA accelerated inline PHY
    - 64T64R Massive MIMO (early access)
    - Enhanced L1-L2 interface
    - 4T4R @ 100MHz multicell capacity on Grace Hopper
    - CSI-P2 enhancement
    - O-RAN Fronthaul
    - Grace Hopper MIG support
    - 4T4T new feature support
- **Aerial cuMAC**: CUDA accelerated MAC scheduler
- **pyAerial**: Python interface to cuPHY modules and pipeline

- **Aerial Data Lake**: Data collection service for PHY to enable AI/ML training

Aerial cuBB

The NVIDIA cuBB SDK provides GPU accelerated 5G signal processing pipeline including cuPHY for Layer 1 PHY, cuMAC for L2 scheduler, delivering unprecedented throughput and efficiency by keeping all the processing within the high-performance GPU memory.

Aerial cuBB is a software-defined, scalable, modular, highly programmable and cloud-native, without any fixed function accelerators. Enables the ecosystem to flexibly adopt necessary modules for their commercial products.

Aerial cuBB has the following key components:

- **cuPHY**: L1 library of the Aerial CUDA-Accelerated RAN. It is designed as an inline accelerator to run on NVIDIA GPUs and it does not require any additional hardware accelerator.
- **cuMAC**: L2 MAC Scheduler library of the Aerial CUDA-Accelerated RAN for accelerating 5G/6G MAC layer scheduler functions with NVIDIA GPUs.

cuBB Installation Guide

This section describes how to install the Aerial cuBB.

Important Terms

Term or Abbreviation	Definition
Aerial	SDK that accelerates 5G RAN functions with NVIDIA GPUs
cuBB	CUDA GPU software libraries/tools that accelerate 5G RAN compute-intensive processing
cuPHY	CUDA 5G PHY layer software library for the cuBB
cuPHY-CP	cuPHY control-plane software

cuMAC	CUDA-based platform for accelerating 5G/6G MAC layer scheduler functions with NVIDIA GPUs
HDF5	A data file format used for storing test vectors. The HDF5 software library provides the functions for reading and writing the test vectors.
CMake	A software tool for configuring the makefiles for building the CUDA examples (see https://cmake.org/)
DPDK	Data Plane Development Kit
CX6-DX	Mellanox ConnectX6-DX NIC

cuBB Quickstart Guide

This section explains how to run the Aerial cuBB software examples.

Important Terms

Term or Abbreviation	Definition
Aerial	Software suite that accelerates 5G RAN functions with NVIDIA GPUs
cuBB	CUDA GPU software libraries/tools that accelerate 5G RAN compute-intensive processing
cuPHY	CUDA 5G PHY layer software library of the cuBB
cuPHY-CP	cuPHY control-plane software
HDF5	A data file format used for storing test vectors. The HDF5 software library provides the functions for reading and writing test vectors.
CMake	A software tool for configuring the makefiles for building the CUDA examples (https://cmake.org/)
DPDK	Data Plane Development Kit
DOCA	DOCA is a software framework that helps developers create applications and services on top of the NVIDIA BlueField networking platform.
GDR	GPUDirect RDMA

FH	Fronthaul
TV	Test Vector

Aerial cuPHY

cuPHY is the 5G L1 library of the Aerial CUDA-Accelerated RAN. It is designed as an inline accelerator to run on NVIDIA GPUs and it does not require any additional hardware accelerator.

Aerial cuPHY Developer Guide

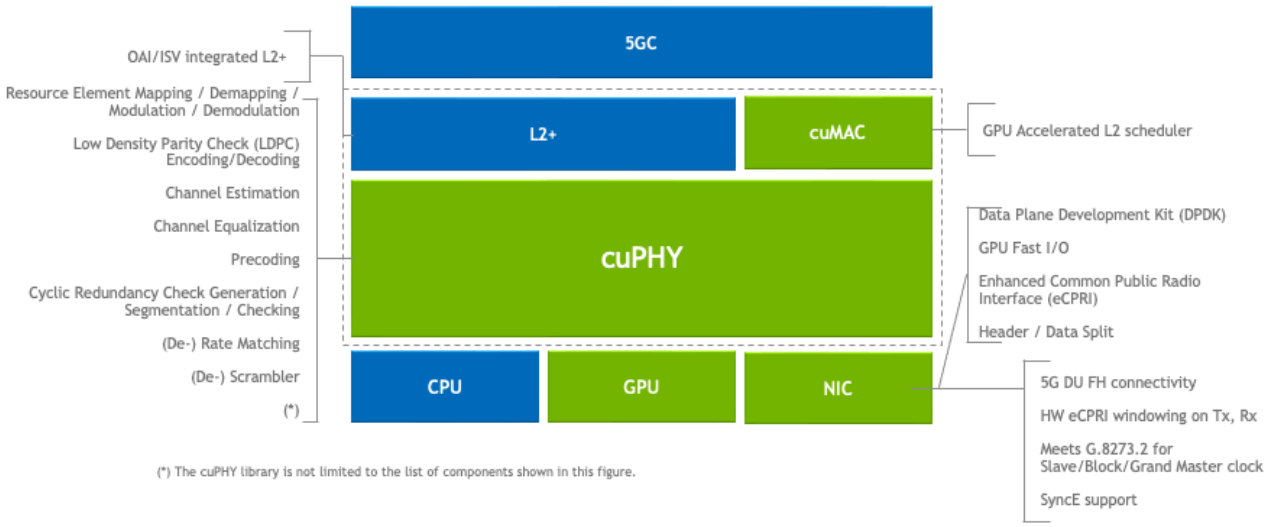
Aerial CUDA-Accelerated RAN is a set of software defined libraries that are optimized to run 5G gNB workloads on GPU. These libraries include cuPHY, cuMAC and pyAerial. In this section, we focus on layer-1 (L1), or physical (PHY) layer of 5G gNB software stack as defined by 3GPP [1-5].

cuPHY is the 5G L1 library of the Aerial CUDA-Accelerated RAN. It is designed as an inline accelerator to run on NVIDIA GPUs and it does not require any additional hardware accelerator. It is implemented according to the O-RAN 7.2 split option [8]. cuPHY library takes advantage of massively parallel GPU architecture to accelerate computationally heavy signal processing tasks. It also makes use of fast GPU I/O interface between the NVIDIA Bluefield-3 (BF3) NIC and GPU (GPU Direct RDMA [7]) to improve the latency.

BF3 NIC provides the fronthaul (FH) connectivity in addition to the IEEE 1588 compliant timing synchronization. The BF3 NIC also has a built-in SyncE and eCPRI windowing functionality, which meets G.8273.2 timing requirements.

In the following, we first give an overview of cuPHY library software stack. cuPHY library consists of L1 controller components running on the CPU and PHY layer functions running on the GPU. After providing the overview, we will go into details of each component and explain how L1 controller components interact with each other and L2.

Finally, we will go over the PHY layer signal processing functions, which are accelerated as CUDA kernel implementations.



Aerial CUDA-Accelerated Software Stack within 5G gNB DU

Acronyms and Definitions

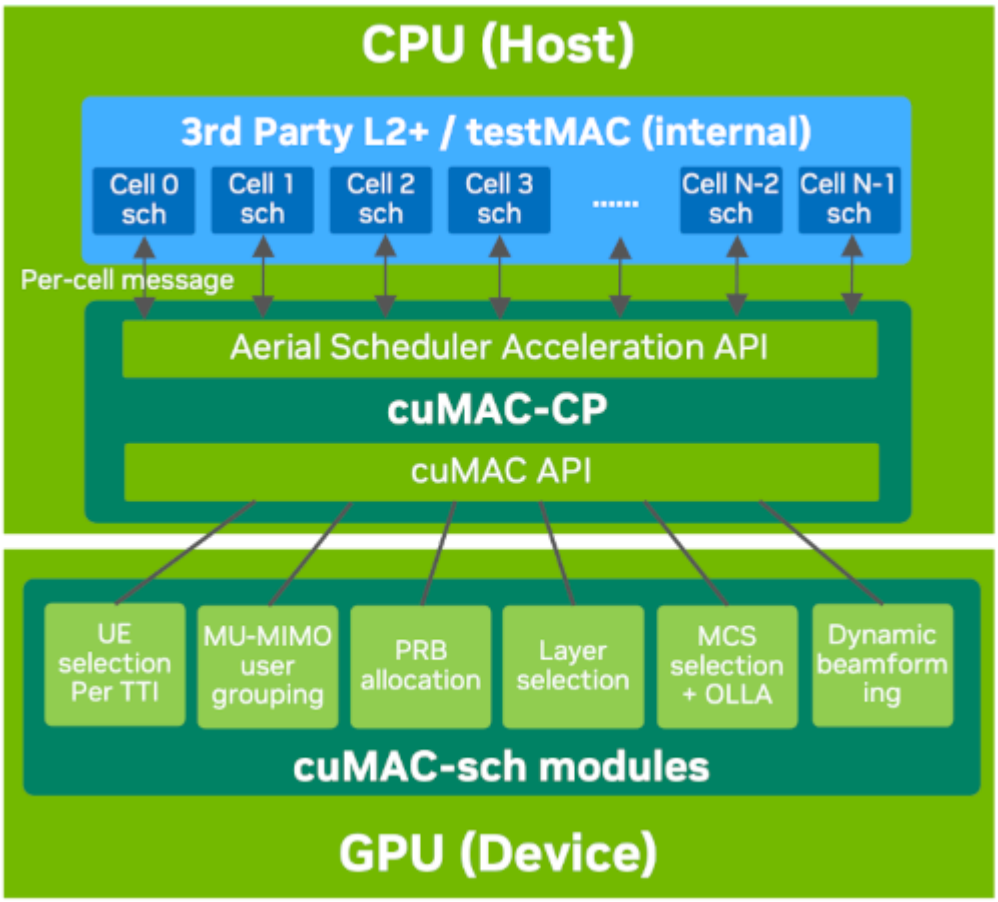
Acronym	Description
3GPP	Third Generation Partnership Project
5G NR	Fifth generation new radio
CB	Code Block
CSI	Channel State Information
CSI-RS	Channel State Information Reference Signal
CUDA	Compute Unified Device Architecture
cuBB	CUDA base-band (L1 software stack consisting of L2 adapter, PHY control layer and PHY layer)
CUDA	Compute Unified Device Architecture
cuPHY	CUDA PHY (L1 functionality on the GPU accelerator in inline mode)
DCI	Downlink Control Information

DL	Downlink
DMRS	Demodulation Reference Signal
DU or O-DU	O-RAN Distributed Unit (a logical node hosting RLC/MAC/High-PHY layers based on a lower layer functional split.)
eCPRI	Ethernet Common Public Radio Interface
eAxC	Extended Antenna Carrier: a data flow for a single antenna (or spatial stream) for a single carrier in a single sector
FAPI	Functional Application Programming Interface
FH	Fronthaul
H2D	Host-to-device memory
LDPC	Low-density Parity Check
NIC	Network interface card
O-RAN	Open RAN
PBCH	Physical Broadcast Channel
PDCCH	Physical Downlink Control Channel
PDSCH	Physical Downlink Shared Channel
PRACH	Physical Random Access Channel
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
RAN	Radio Access Network
RM	Reed-Muller
RU or O-RU	O-RAN Radio Unit: a logical node hosting Low-PHY layer and RF processing based on a lower layer functional split
SCF	Small Cell Forum
SSB	Synchronization Signal Block

SyncE	Synchronous Ethernet: is an ITU-T standard to provide a synchronization signal to network resources
UCI	Uplink Control Information
UL	Uplink
TB	Transport Block

Aerial cuMAC

Aerial cuMAC is a CUDA-based platform for accelerating 5G/6G MAC layer scheduler functions with NVIDIA GPUs. cuMAC supported scheduler functions include UE selection/grouping, PRB allocation, layer selection, MCS selection/link adaptation and dynamic beamforming, all designed for the joint scheduling of multiple coordinated cells. cuMAC offers a C/C++ based API for the offloading of scheduler functions from the L2 stack in the DUs to GPUs. In the future, cuMAC will evolve into a platform that combines AI/ML based scheduler enhancements with GPU acceleration.



Aerial L2 scheduler acceleration data flow chart

cuMAC is the main component of the Aerial L2 scheduler acceleration solution. The figure above illustrates the overall data flow of the scheduler acceleration. The full solution consists of the following components: 1) Aerial Scheduler Acceleration API, which is a per-cell message passing-based interface between the 3rd party L2 stack on DU/CU and cuMAC-CP, 2) cuMAC-CP, 3) cell group-based cuMAC API, and 4)cuMAC multi-cell scheduler (cuMAC-sch) modules.

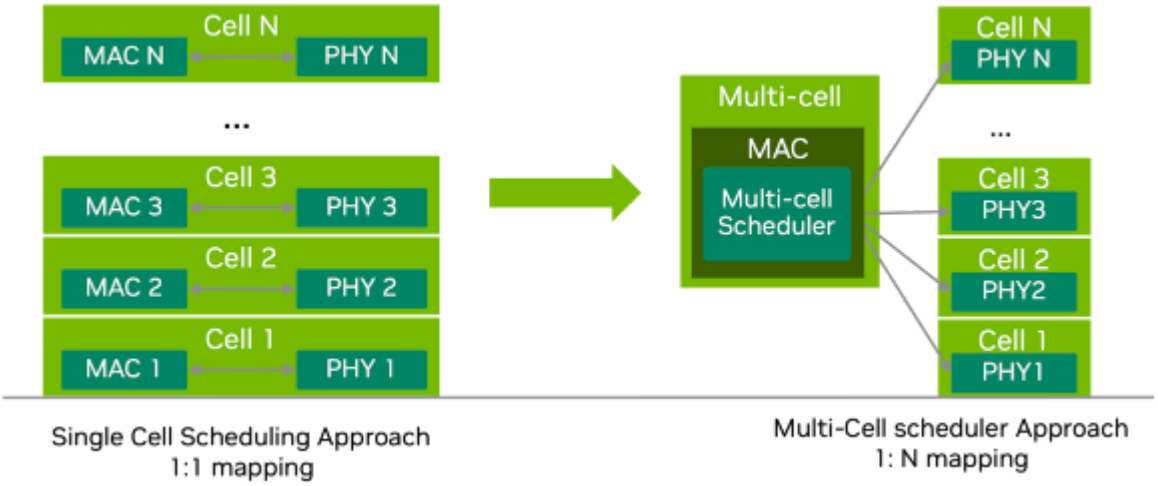
The 3rd party L2 stack sits on the CPU and contains a single-cell L2 scheduler for each individual cell under its control. To offload L2 scheduling to GPU for acceleration/performance purposes, in each time slot (TTI), the L2 stack host sends per-cell request messages to cuMAC-CP through the Aerial Scheduler Acceleration API, which consists of required scheduling input & config. information from each single-cell scheduler. Upon receiving the per-cell request messages, cuMAC-CP integrates all scheduler input information from those (coordinated) cells into the cuMAC API cell group data structures and populates the GPU data buffers contained in these structures. Next, the cuMAC multi-cell scheduler (cuMAC-sch) modules are called by cuMAC-CP through cuMAC API to compute scheduling solutions for the given time slot (TTI). After the cuMAC-

sch modules complete the computation and the scheduling solutions become available in the GPU memory, cuMAC-CP converts them into per-cell response messages and sends them back to the L2 stack host on CPU through the Aerial Scheduler Acceleration API. Finally, the L2 stack host uses the obtained solutions to schedule the cells under its control.

When there are multiple coordinated cell groups, a separate set of Aerial Scheduler Acceleration API, cuMAC-CP, cuMAC API and cuMAC instances should be constructed and maintained for each cell group.

Implementation Details

- **Multi-cell scheduling -** All cuMAC scheduling algorithms are implemented as CUDA kernels that are executed by GPU and jointly compute the scheduling solutions (PRB allocation, MCS selection, layer selection, etc.) for a group of cells at the same time. The algorithms can be constrained to single cell scheduling by configuring a single cell in the cell group. A comparison between the single-cell scheduler and multi-cell scheduler approaches is given in the below figure.



Single-cell scheduler approach vs. multi-cell scheduler approach

- **Scheduling algorithm CUDA implementation**
    - **PF UE down-selection algorithm -** cuMAC offers a PF-based UE selection algorithm to down-select a subset of UEs for new transmissions or HARQ re-transmissions in each TTI from the pool of all active UEs in each cell of a cell group. The association of UEs and cells in the cell group is an input to the UE selection module. When selecting UEs for each cell in each TTI, the UE selection algorithm first assigns a priority weight to each active UE in a cell and

then sorts all active UEs in descending order of the priority weight. The subset of UEs that have the highest priority weights in each cell are selected for scheduling in a TTI. The number of selected UEs per cell is an input parameter to this module. HARQ re-transmissions are always assigned with the highest priority weight. For the new-transmission UEs, their priority weights are the PF metrics, calculated as the ratio of each UE's long-term average throughput and its instantaneous achievable data rate. The UE selection algorithm is implemented as CUDA kernels that run on GPU and jointly select UEs for all cells in a cell group at the same time.

- **PF PRB allocation algorithms -** cuMAC offers algorithms to perform channel-aware and frequency-selective PRB allocation among a group of cells and their connected active UEs on a per-TTI basis. The input arguments to the PRB allocation algorithms include the narrow-band SRS channel estimates (MIMO channel matrices) per cell-UE link, the association solutions between cells and UEs, and other UE status and cell group parameters. The output is the PRB allocation solution for the cell group, whose data format depends on the type of allocation: 1) for type-0 allocation, a per UE binary bitmap indicating whether each PRB is allocated to the UE, and 2) for type-1 allocation, with 2 elements per UE indicating the starting and ending PRB indices for the UE's allocation. Two versions of the PRB allocation algorithms are provided, one for single cell scheduling and the other for multi-cell joint scheduling. A major difference between the two versions is that the multi-cell algorithm considers the impact of inter-cell interference in the evaluation of per-PRB SINRs, which can be derived from the narrow-band SRS channel estimates. The single-cell version does not explicitly consider inter-cell interference and only utilizes information restricted to each individual cell. The multi-cell algorithm can lead to a globally optimized resource allocation in a cell group by leveraging all available information from the coordinated multiple cells. A prototyping CUDA kernel implementation of PRB allocation algorithms is provided in the figure below.

- **Layer selection algorithm -** cuMAC offers layer selection algorithms that choose the best set of layers for transmission for a UE based on the singular value distribution across the UE's multiple layers. A predetermined singular value threshold is used to find the number of layers (with descending singular values) that can be supported on each subband (PRB group). Then the minimum number of layers across all allocated subbands to the UE is chosen as the optimal layer selection solution. Input arguments to the layer selection algorithms include the PRB allocation solution per UE, the singular values of each UE's channel on its allocated subbands, the association solutions

between cells and UEs, and other UE status and cell group parameters. The output is the per-UE layer selection solution. The layer selection algorithm is implemented as CUDA kernels that run on GPU and jointly select layers for all UEs in a cell group at the same time.

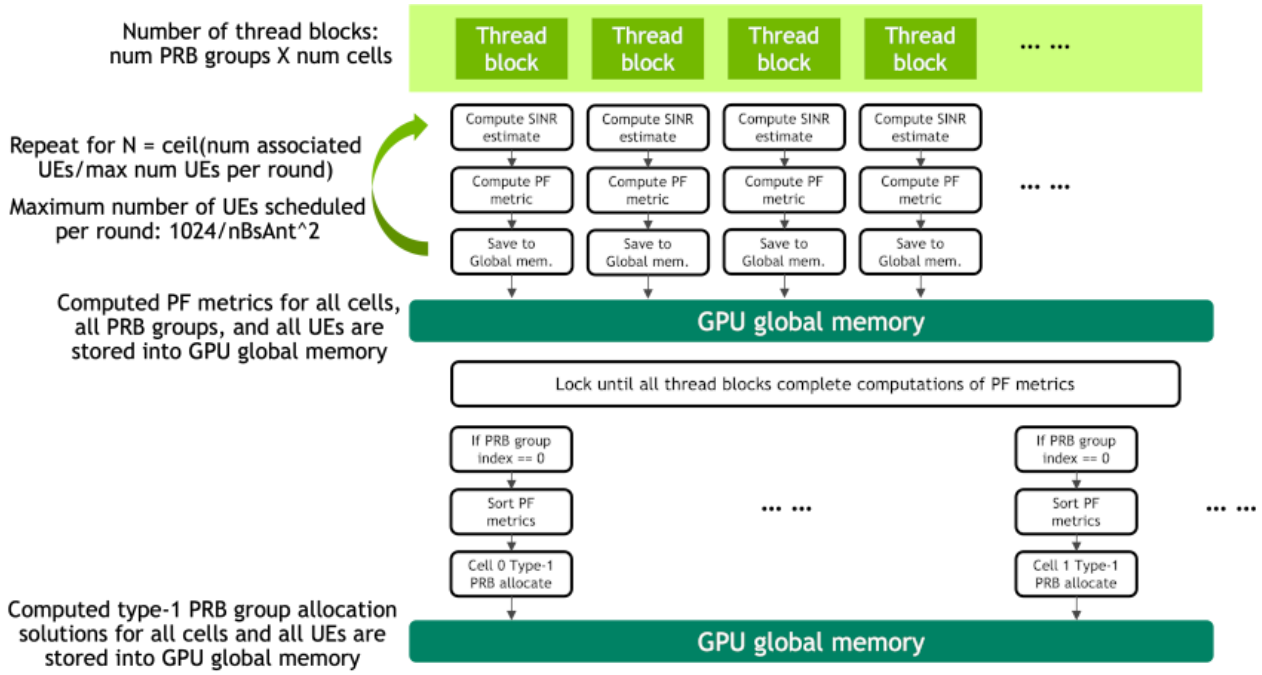
  - **MCS selection algorithm -** cuMAC offers MCS selection algorithms that choose the best feasible MCS (highest level that can meet a given BLER target) per UE based on a given PRB allocation solution. An outer-loop link adaptation algorithm is integrated internally to the MCS selection algorithm, which offsets the SINR estimates based on previous transport block decoding results per UE link. Input arguments to the MCS selection algorithms include the PRB allocation solution per UE, the narrow-band SRS channel estimates (MIMO channel matrices) per cell-UE link, the association solutions between cells and UEs, the decoding results of the last transport block for each UE, and other UE status and cell group parameters. The output is the per-UE MCS selection solution. The MCS selection algorithm is implemented as CUDA kernels that run on GPU and jointly select MCS for all UEs in a cell group at the same time.

  - **Support for HARQ -** all the above cuMAC scheduler algorithms can support HARQ re-transmissions with non-adaptative mode, i.e., reusing the same scheduling solution of the initial transmission for re-transmissions.

- **CPU reference code -** CPU C++ implementation of the above algorithms is also provided for verification and performance evaluation purposes.

- **Different CSI types -** cuMAC offers scheduler algorithm CUDA kernels to work with different CSI types, including SRS channel coefficient estimates and CSI-RS based channel quality information.

- **Support for FP32 and FP16 -** cuMAC offers scheduler algorithm CUDA kernels implemented in FP32 and FP16. Using FP16 kernels can help reduce scheduler latency with a minor performance loss.



A prototyping CUDA kernel implementation of PRB allocation algorithms

Aerial Data Lake

6G will be artificial intelligence (AI) native. AI and machine learning (ML) will extend through all aspects of next generation networks from the radio, baseband processing, the network core including system management, orchestration and dynamic optimization processes. GPU hardware, together with programming frameworks will be essential to realize this vision of a software defined native-AI communication infrastructure.

The application of AI/ML in the physical layer has particularly been a hot research topic.

There is no AI without data. While the synthetic data generation capabilities of Aerial Omniverse Digital Twin (AODT) and Sionna/SionnaRT are essential aspects of a research project, availability of over-the-air (OTA) waveform data from real-time systems is equally important. This is the role of Aerial Data Lake. It is a data capture platform supporting the capture of OTA radio frequency (RF) data from virtual radio access network (vRAN) networks built on the Aerial CUDA-Accelerated RAN. Aerial Data Lake consists of a data capture application (app) running on the base station (BS) distributed unit (DU), a database of samples collected by the app, and an application programming interface (API) for accessing the database.

Target Audience

Industry and university researchers and developers looking to bring ML to the physical layer with the end goal of benchmarking on OTA testbeds like NVIDIA ARC-OTA or other GPU-based BSs.

Key Features

Aerial Data Lake has the following features:

Real-time capture of RF data from OTA testbed

- Aerial Data Lake is designed to operate with gnBs built on the Aerial CUDA-Accelerated RAN and that employ the Small Cell Forum FAPI interface between L2 and L1. One example system being the NVIDIA ARC-OTA network testbed. I/Q samples from O-RUs connected to the GPU platform via a O-RAN 7.2x split fronthaul

interface are delivered to the host CPU and exported to the Aerial Data Lake database.

Aerial Data Lake APIs to access the RF database

- The data passed to the layer-2 via RX_Data.Indication and UL_TTI.Request are exported to the database. The fields in these data structures form the basis of the database access APIs.

Scalable and time coherent over arbitrary number of BSs

- The data collection app runs on the same CPU that supports the DU. It runs on a single core, and the database runs on free cores. Because each BS is responsible for collecting its own uplink data, the collection process scales as more BSs are added to the network testbed. Database entires are time-stamped so data collected over multiple BSs can be used in a training flow in a time-coherent manner.

Use in conjunction with pyAerial to generate training data for neural network physical layer designs

- Aerial Data Lake can be used in conjunction with the NVIDIA pyAerial CUDA-Accelerated Python L1 library. Using the Data Lake database APIs, pyAerial can access RF samples in a Data Lake database and transform those samples into training data for all the signal processing functions in an uplink or downlink pipeline.

Design

Aerial Data Lake sits beside the Aerial L1 and copies out data that would be useful for machine learning into an external database.

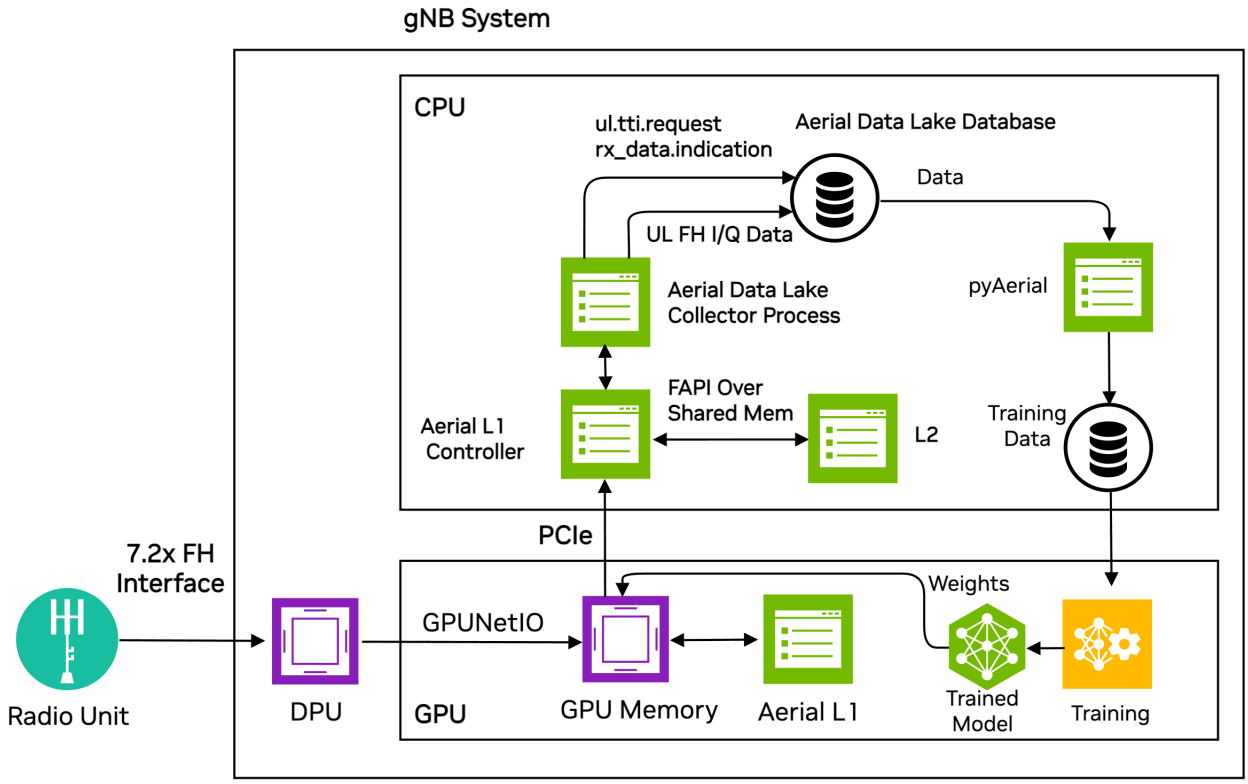


Figure 1: The Aerial Data Lake data capture platform as part of the gNB.

Uplink I/Q data from one or more O-RAN radio units (O-RUs) is delivered to GPU memory where it is both processed by the Aerial L1 PUSCH baseband pipeline and delivered to host CPU memory. The Aerial Data Lake collector process writes the I/Q samples to the Aerial Data Lake database in the *fh* table. The *fh* table has columns for SFN, Slot, IQ samples as *fhData*, and the start time of that SFN.slot as *TsTaiNs*.

The collector app saves data that the L2 sent to L1 to describe UL OTA transmissions in UL_TTI.Request messages as well as data returned to L2 the via RX_Data.Indication and CRC.Indication. This data is then written to the *fapi* database table. These messages and the fields within them are described in SCF 5G FAPI PHY Spec version 10.02, sections 3.4.3, 3.4.7, and 3.4.8.

Each gNB in a network testbed collects data from all O-RUs associated with it. That is, data collection over the span of a network is performed in a distributed manner, each gNB is building its own local database. Training can be performed locally at each gNB, and site-specific optimizations can be realized with this approach. Since the data in a database is time-stamped, the local databases can be consolidated at a centralized compute resource and training performed using the time aligned aggregated data. In cases where the aerial pusch pipeline was unable to decode due to channel conditions,

retransmissions can be used as ground truth as long as one of the retransmissions succeeds, allowing the user to test algorithms with better performance than the originals.

The Aerial Data Lake database storage requirements depend on the number of O-RUs, the antenna configuration of the O-RU, the carrier bandwidth, the TDD pattern and the number of samples to be collected. Collecting IQ samples of 1 million tranmissions from a single RU 4T4R O-RU employing a single 100MHz carrier will consume approximately 660 GB of storage.

Aerial Data Lake database comprises the fronthaul RF data. However, for many training applications access to data at other nodes in the receive pipeline is required. A pyAerial pipeline, together with the Data Lake database APIs, can access samples from an Aerial Data Lake database and transform that data into training data for any function in the pipeline.

Figure 2 illustrates data ingress from a Data Lake database into a pyAerial pipeline and using standard Python file I/O to generate training data for a soft de-mapper.

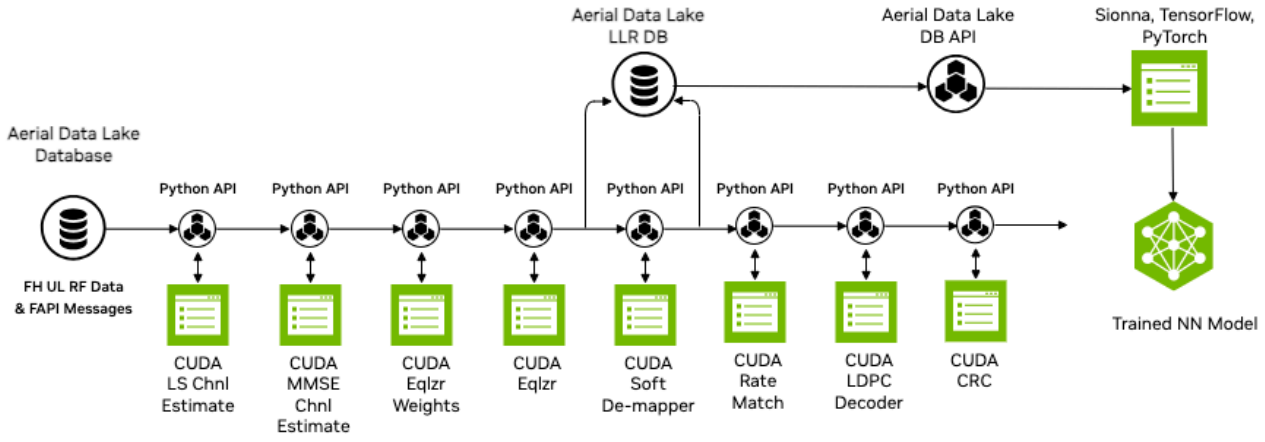


Figure 2: pyAerial is used in conjunction with the NVIDIA data collection platform, namely, Aerial Data Lake to build training data sets for any node in the layer-1 downlink or uplink signal processing pipeline. The example shows a Data Lake database of over-the-air samples transformed into training data for a neural network soft de-mapper.

Installation

Aerial Data Lake is compiled by default as part of cuphycontoller. If you would like to record fresh data every time cuphycontroller is started, see the section on Fresh Data.

Start by installing Clickhouse database on the server collecting the data. The command below will download and run an instance of the clickhouse server in a docker container.

```
docker run -d \ --network=host \ -v $(realpath ./ch_data):/var/lib/clickhouse/ \ -v $(realpath ./ch_logs):/var/log/clickhouse-server/ \ --cap-add=SYS_NICE --cap-add=NET_ADMIN --cap-add=IPC_LOCK \ --name my-clickhouse-server --ulimit nofile=262144:262144 clickhouse/clickhouse-server
```

By default clickhouse will not drop large tables, and will return an error if attempted. The clickhouse-cpp library does not return exceptions so to avoid what loooks like a cuphycontroller crash we recommend allowing it to drop large tables using the following command:

```
sudo touch './ch_data/flags/force_drop_table' && sudo chmod 666 './ch_data/flags/force_drop_table'
```

Usage

In the cuphycontoller adapter yaml configuration file, enable data collection by specifying a core then start cuphycontroller and usual. The core should be on the same NUMA node as the rest of cuphycontroller, i.e. should follow the same pattern as the rest of the cores An example of this can be found commented out in cuphycontroller_P5G_FXN_R750.yaml.

```
cuphydriver_config: # Fields added for data collection datalake_core: 19 # Core on which data collection runs. E.g isolated odd on R750, any isolated core on gigabyte datalake_address: localhost datalake_samples: 1000000 # Number of samples to collect for each UE/RNTI. Defaults to 1M
```

When enabled the *DataLake* object is created and *DataLake::dbInit()* initializes the two tables in the database. After cuphycontroller runs the PUSCH pipeline, cupycontroller calls *DataLake::notify()* with the addresses of the data to be saved, which *DataLake* then saves. When *DataLake::waitForLakeData* wakes up it calls *DataLake::dbInsert()* which appends data to respective *Clickhouse* columns, then sleeps waiting for more data. Once 20 PUSCH transmissions have been stored or a total of *datalake_samples* have been recived the columns are appended to a *Clickhouse::Block* and inserted into the respective table.

Using Data Lake in Notebooks

Follow pyAerial instructions and usual to build and launch that container. It must be run on a server with a GPU.

Two example notebooks for are included:
datalake_channel_estimation.ipynb performs channel estimation and plots the result
datalake_chan_estimation_decoding.ipynb goes futher and runs the full PUSCH decoding pipeline, both a fused version and a version build up up constituent parts

Notes

Database Administration

Clickhouse client

A clickhouse client is needed to interact with the server. To download it and run it do the following:

> curl https://clickhouse.com/ | sh ./clickhouse client aerial@aerial-gnb:~$ ./clickhouse client ClickHouse client version 24.3.1.1159 (official build). Connecting to localhost:9000 as user default. Connected to ClickHouse server version 24.3.1. aerial-gnb :)

You are now at the clickhouse client prompt. Commands starting with *aerial-gnb :)* are entered at this prompt and those with *$* are run on the host.

Database Import

There are example *fapi* and *fh* tables included in Aerial CUDA-Accelerated RAN. These tables can be imported into the clickhouse database by copying them to the clickhouse user_files folder, using the client to import them:

> $ docker cp c_aerial_$USER:/opt/nvidia/cuBB/pyaerial/notebooks/data/fh.parquet . $ docker cp c_aerial_$USER:/opt/nvidia/cuBB/pyaerial/notebooks/data/fapi.parquet . $ sudo cp *.parquet ./ch_data/user_files/ aerial-gnb :) create table fapi ENGINE =

```
MergeTree primary key TsTaiNs settings allow_nullable_key=1 as select * from file('fapi.parquet',Parquet) Ok. aerial-gnb :) create table fh ENGINE = MergeTree primary key TsTaiNs settings allow_nullable_key=1 as select * from file('fh.parquet',Parquet) Ok. aerial-gnb :) select table, formatReadableSize(sum(bytes)) as size from system.parts group by table SELECT `table`, formatReadableSize(sum(bytes)) AS size FROM system.parts GROUP BY `table` Query id: 95451ea7-6ea9-4eec-b297-15de78036ada
   table                    size          fh    4.54 MiB      fapi    2.18 KiB
```

You now have five PUSCH transmissions loaded in the database and can run the example notebooks.

Database Queries

To show some information about the entries (rows) you can run the following:

```
# Show counts of transmissions for all RNTIs aerial-gnb :) select rnti, count(*) from fapi group by rnti SELECT rnti, count(*) FROM fapi GROUP BY rnti Query id: 76cf63d8-7302-4d73-972e-8ba7392da7ac      rnti      count()       55581    5
                              # Show select information from all rows of the fapi table aerial-gnb :) from fapi select TsTaiNs,TsSwNs,SFN,Slot,pduData SELECT TsTaiNs, TsSwNs, SFN, Slot, pduData FROM fapi Query id: af362836-b379-46fd-85ae-0e9f62deb8ab
                                    TsTaiNs                                       TsSwNs      SF
  2024-03-21 12:18:39.162000000    2024-03-21 12:18:39.162990534    192    4
[62,1,0,63,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,
    2024-03-21 12:18:39.187000000    2024-03-21 12:18:39.188086009    194    14
[1,36,192,1,0,1,58,12,191,0,166,41,62,128,2,191,0,46,0,6,128,128,120,136,120,127,128,
    2024-03-21 12:18:39.192000000    2024-03-21 12:18:39.194784691    195    4
[62,1,0,63,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,
    2024-03-21 12:18:39.252000000    2024-03-21 12:18:39.253086195    201    4
[1,3,0,2,0,61,0,57,63,51,63,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33,33
    2024-03-21 12:18:39.332000000    2024-03-21 12:18:39.332997301    209    4
[1,38,192,2,0,2,58,13,191,0,216,239,96,3,131,63,0,44,138,152,7,112,128,220,160,94,152
```

```
5 rows in set. Elapsed: 0.002 sec. #Show start times of fh table aerial-gnb :) from fh select TsTaiNs,TsSwNs,SFN,Slot SELECT TsTaiNs, TsSwNs, SFN, Slot FROM fh Query id: 078d451a-5db9-4f35-b890-96b2c561fdbe
                                TsTaiNs                                 TsSwNs      SF
  2024-03-21 12:18:39.162000000    2024-03-21 12:18:39.162990534    192    4
2024-03-21 12:18:39.187000000    2024-03-21 12:18:39.188086009    194    14
2024-03-21 12:18:39.192000000    2024-03-21 12:18:39.194784691    195    4
2024-03-21 12:18:39.252000000    2024-03-21 12:18:39.253086195    201    4
2024-03-21 12:18:39.332000000    2024-03-21 12:18:39.332997301    209    4

5 rows in set. Elapsed: 0.002 sec.
```

Fresh Data

The database of IQ samples grows quite quickly. If you want fresh data every run the tables can be dropped automatically by uncommenting these lines in cuPHY-CP/data_lakes/data_lakes.cpp:

```
//dbClient->Execute("DROP TABLE IF EXISTS fapi"); //dbClient->Execute("DROP TABLE IF EXISTS fh");
```

Dropping Data

You can manually drop all of the data from the database with these commands:

```
aerial-gnb :) drop table fh Ok. aerial-gnb :) drop table fapi Ok.
```

Jupyter notebooks

Exceptions are not always displayed in jupyter notebooks the way that it would be if a python script had been run, so in some cases it can be easier to convert the notebook to a script and run that.

```
jupyter nbconvert --to script <notebook_name>.ipynb
```

To interact with the data and code in place, specific lines can be debugged by adding *breakpoint()* inline

Known Limitations

Currently datalakes records the first UE per TTI and has been tested with a single cell per gNB as supported by the Open Air Interface L2+ stack.

pyAerial

As 6G research gains momentum, and with many new technologies in its purvue, one thing is clear, AI/ML will feature prominently in the next generation RAN. It will play a pivotal role in realizing all parts of the network infrastructure from the radio units, baseband processing, the network core including system management, orchestration and dynamic optimization processes. GPU hardware, together with programming frameworks will be essential to realize this vision of a software defined native-AI communication infrastructure.

The application of AI/ML in the physical layer has in particular been a hot research topic. There is a lot of emphasis on neural network architectures and optimization strategies mostly performed in the context of simulation. The next step for the research community and commercial system developers is to bring AI/ML applied in layer-1 to reality in over-the-air real-time testbeds and operator-network scale systems.

This is where pyAerial enters the picture. pyAerial is a Python library of physical layer components that can be used as part of the workflow in taking a design from simulation to real-time operation. It helps with end-to-end verification of a neural network integration into a PHY pipeline and helps bridge the gap from the world of training and simulation in TensorFlow/PyTorch to real-time operation in an over-the-air testbed.

The pyAerial library provides a Python-callable bit-accurate GPU-accelerated library for all of the signal processing CUDA kernels in the NVIDIA cuBB layer-1 PDSCH and PUSCH pipelines. In other words, the pyAerial Python classes behave in a numerically identical manner to the kernels employed in cuBB because a pyAerial class employs the exact same CUDA code as the corresponding cuBB kernel: it is the CUDA kernel but with a Python API.

Using pyAerial library components complete layer-1 pipelines can be composed in Python. User code or inference engines, from NVIDIA TensorRT, or custom CUDA code, can be included in the datapath as shown in the lower part of Figure 1. This rapid prototyping design and verification flow is used for dataplane functional performance evaluation. It is a step in the workflow for verifying a physical layer design prior to deployment in a real-time over-the-air GPU base station.

pyAerial can also be used in conjunction with the NVIDIA data collection platform *Aerial Data Lake.* An Aerial Data Lake database consists of RF samples from a 7.2x fronthaul interface together with L2 meta-information to enable database search and query operations. A pyAerial pipeline can access samples from Aerial Data Lake database using the Data Lake Python APIs, and transform that data into training data for any function in the pipeline. Figure 2 illustrates data ingress from a Data Lake database into a pyAerial pipeline and using standard Python file I/O to generate training data for a soft de-mapper.

Content

Key Features

pyAerial has the following key features:

Feature 1: Productive Python for rapid prototyping of layer-1 pipelines

- pyAerial library components are CUDA kernels with Python bindings. The productive environment of Python permits the rapid assembly of signal processing pipelines in

Python. All of the analytic and visualization aspects of Python can be used for performance characterization, signal visualization and debugging.

Feature 2: Simulate machine learning in the physical layer before over-the-air operation

- With the goal of going from model training and simulation in TensorFlow or PyTorch to real-time over-the-air operation, pyAerial provides a convenient way to verify, evaluate and benchmark your physical layer prior to deployment in an OTA testbed.

Feature 3: Fast simulation with CUDA optimized kernels

- pyAerial library components are CUDA under the hood. Simulation is fast on a GPU. When you are simulating the coding chain, including for example an LDPC decoder, optimized CUDA code is implementing these computationally heavy functions.

Feature 4: Generate data sets for any node in layer-1 uplink or downlink pipeline

- pyAerial is designed to be used in conjunction with the NVIDIA data collection platform *Aerial Data Lake*. pyAerial can access RF samples in a Data Lake database and transform those samples into training data for all of the signal processing functions in and uplink or downlink pipeline.

Feature 5: Bit accurate simulation

- Because pyAerial is Python running on CUDA, the performance you observe in BLER and other characterization metrics is what is identical to the performance of the real-time over-the-air system.

Target Audience

Industry and university researchers and developers looking to bring machine learning to the physical layer with the end goal of benchmarking on over-the-air testbeds like NVIDIA ARC-OTA or other GPU-based base stations.

Value Proposition

Fast bit-accurate GPU accelerated simulation of neural-network downlink and uplink signal processing pipelines. Rapid prototyping and functional verification of a real-time layer-1 in preparation for real-time deployment. Convenient Python environment aids debugging and provides easy access to all nodes in the pipeline for visualization and analysis. Easy to use Python environment for producing BLER and other statistics of interest for a real-time bit-accurate GPU layer-1 implementation. Transform RF sample captures for over-the-air captures into data for training layer-1 functions or compositions of multiple functions.

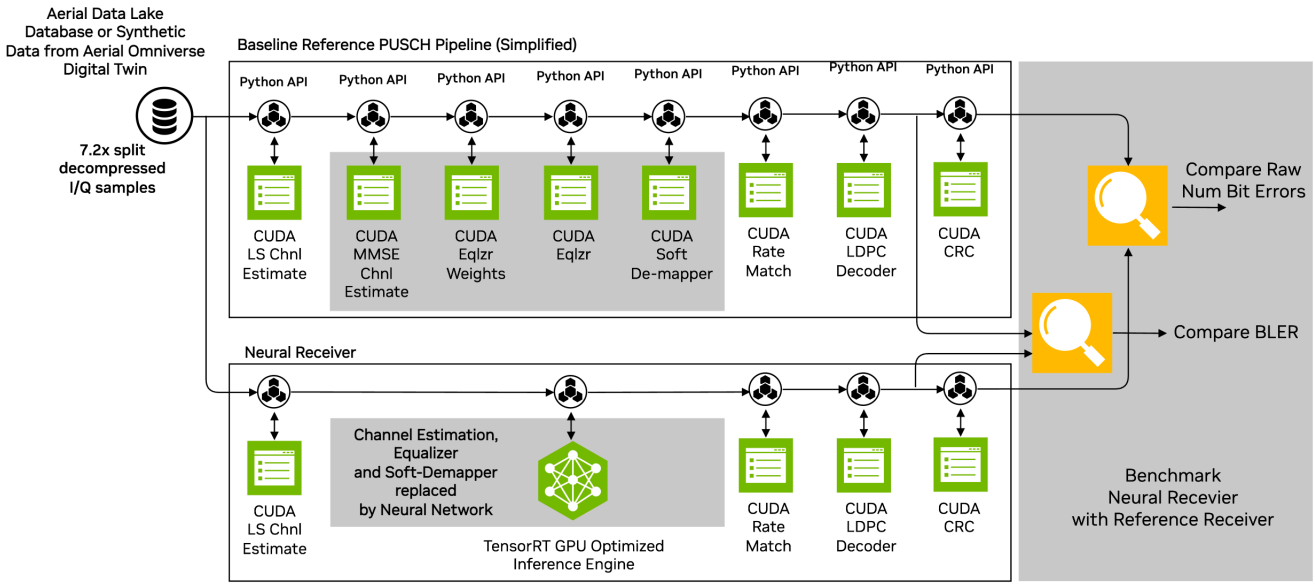


Figure 1: Using pyAerial to verify a neural pipeline context of a full uplink pipeline. This is one of the verification steps to moving to real-time operation over-the-air on a GPU base station.

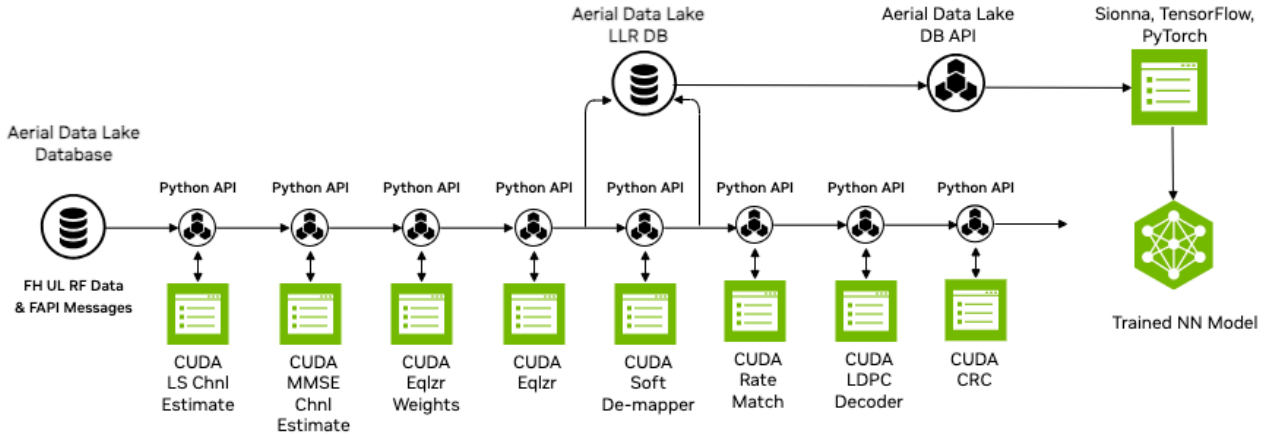


Figure 2: pyAerial is used in conjunction with the NVIDIA data collection platform Aerial Data Lake to build training data sets for any node in the layer-1 downlink or uplink signal processing pipeline. The example shows a Data Lake database of over-the-air samples transformed into training data for a neural network soft de-mapper, using pyAerial. Data gets extracted at the input and output of the de-mapper, and stored in the database.

Release Notes

- Release version: 24-1
- Supported configurations:
    - AX800, A100X and A100 GPUs with the x86 platform.
        - CUDA Toolkit: 12.2.0
        - GPU Driver (OpenRM): 535.54.03
    - Note: The Grace Hopper platform is currently not supported.
- Supported features: pyAerial exposes a subset of the cuPHY API features to Python. Currently this subset includes the following features:
    - PUSCH receiver pipeline

    - PDSCH transmission pipeline
    - Channel estimation
    - Noise and interference estimation
    - Channel equalization and soft demapping
    - LDPC encoding
    - LDPC decoding
    - LDPC rate matching
    - SRS channel estimation

- Limitations:
    - Unlike the cuPHY API, pyAerial API supports only a single UE group per method call. Multiple UE groups (FDM) can be supported by calling the methods separately for each UE group.