Aerial cuPHY Components

Table of contents

List of Figures

L2 Adapter

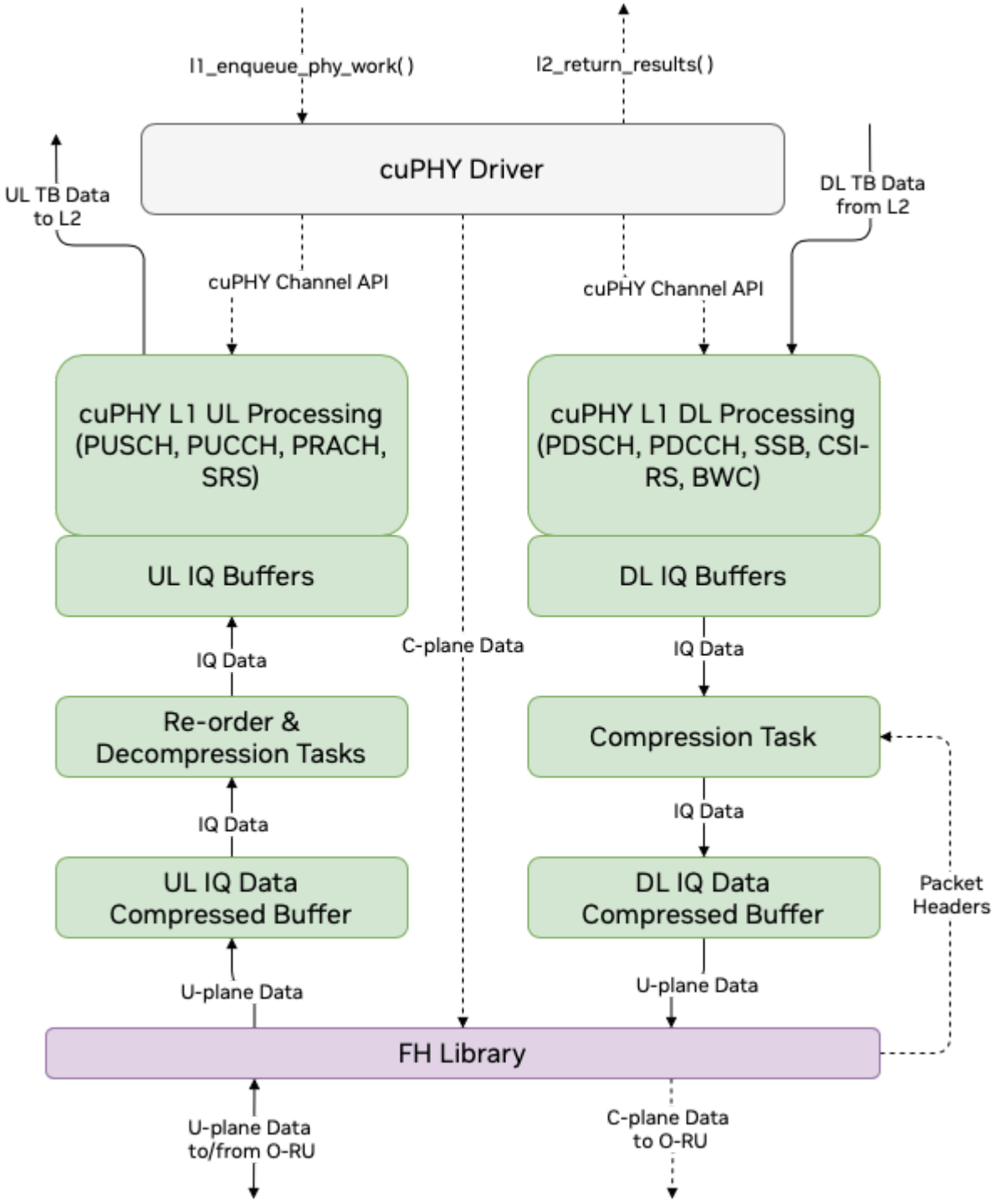
The L2 Adapter is the interface between the L1 and the L2, which translates SCF FAPI commands to slot commands. The slot commands are received by cuPHY driver to initiate cuPHY tasks. It makes use of nvipc library to transport messages and data between L1 and L2. It is also responsible for sending slot indications to drive the timing of the L1-L2 interface. L2 Adapter keeps track of the slot timing and it can drop messages received from L2 if they are received late.

cuPHY Driver

The cuPHY driver is responsible for orchestrating the work on the GPU and the FH by using cuPHY and FH libraries. It processes L2 slot commands generated by L2 adapter to launch tasks and communicates cuPHY outputs (e.g. CRC indication, UCI indication, measurement reports, etc.) back to L2. It uses L2 adapter FAPI message handler library to communicate with L2.

cuPHY driver configures and initiates DL and UL cuPHY tasks, which in turn launch CUDA kernels on the GPU. These processes are managed at the slot level. The cuPHY driver also controls CUDA kernels responsible for transmission and reception of user plane (U-plane) packets to and from the NIC interface. The CUDA kernels launched by the driver take care of re-ordering and decompression of UL packets and compression of DL packets. The DL packets are transmitted by GPU initiated communications after the compression.

cuPHY driver interacts with the FH interface using ORAN compliant FH library to coordinate transmission of FH control plane (C-plane) packets. The transmission of C-plane packets is done via DPDK library calls (CPU initiated communication). The U-plane packets are communicated through transmit and receive queues created by the cuphycontroller.



User and Control Plane Data Flow through cuPHY driver and cuPHY tasks

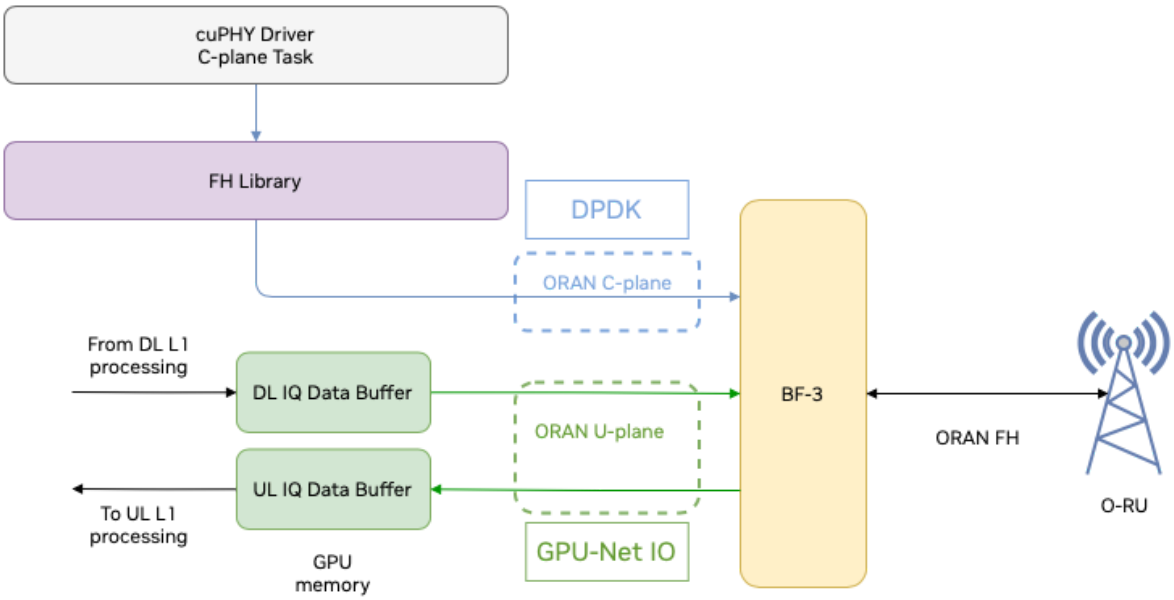
FH Driver Library

The FH library ensures timely transmission and reception of FH packets between the O-DU and O-RU. It uses accurate send scheduling functions of the NIC to comply with the

timing requirements of the O-RAN FH specification.

The FH driver maintains the context and connection per eAxCid. It is responsible of encoding and decoding of FH commands for U-plane and C-plane messages.

The FAPI commands received from the L2 trigger processing of DL or UL slots. C-plane messages are for both DL and UL generated on the CPU and communicated to the O-RU through the NIC interface with DPDK. The payload of DL U-plane packets are prepared on the GPU and sent to the NIC interface from the memory pool on the GPU with the DOCA GPU NetIO library. The flow of DL C-plane and U-plane packets is illustrated in the below figure.



Flow of packets on the FH

As shown in the above figure, UL U-plane packets received from the O-RU are directly copied to GPU memory from the NIC interface with the DOCA GPU NetIO library. The UL data is decompressed and processed by GPU kernels. After the UL kernels are completed, the decoded UL data transport blocks are sent to the L2.

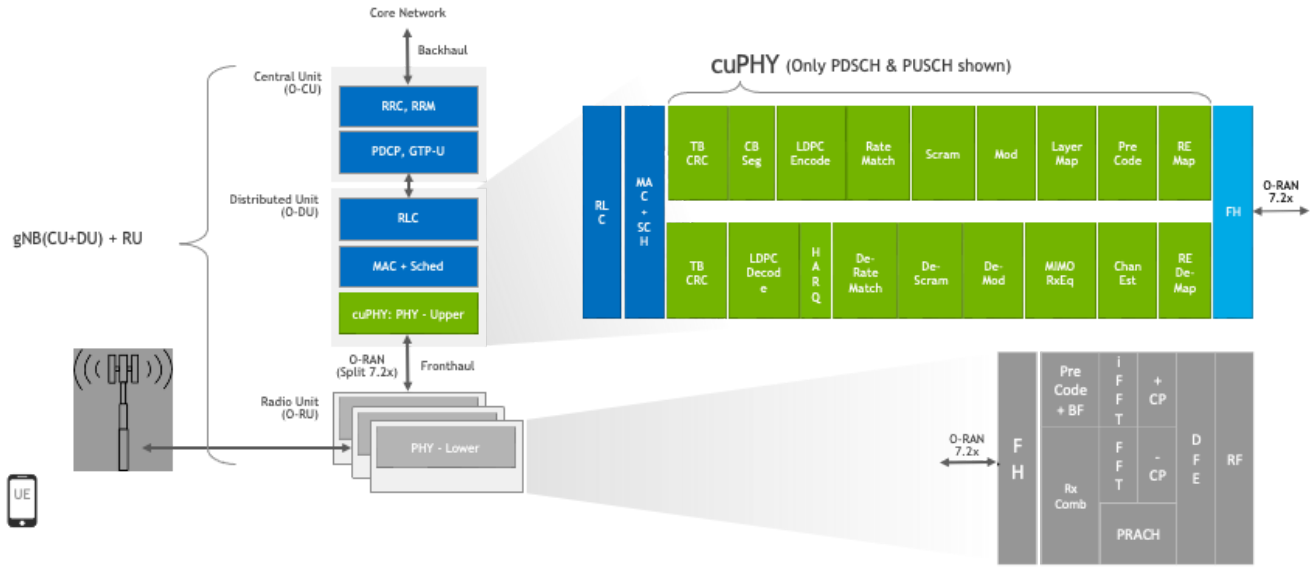
cuPHY Controller

The cuPHY controller is the main application that initializes the system with the desired configuration. During the start-up process, cuPHY controller creates a new context (memory resources, tasks) for each new connection with a O-RU, identified by MAC address, VLAN ID and set of eAxCids. It starts cuphydriver DL/UL worker threads and assigns them to CPU cores as configured in the yaml file. It also prepares GPU resources and initiates FH driver and NIC class objects.

cuPHY controller prepares L1 according to the desired gNB configuration. It can also bring a carrier in and out of service with the cell lifecycle management functionality.

cuPHY

cuPHY is a CUDA implementation of 5G PHY layer signal processing functions. The cuPHY library supports all 5G NR PHY channels in compliance with 3GPP Release 15 specification. As shown in the below figure, cuPHY library corresponds to upper PHY stack according to O-RAN 7.2x split option [8].

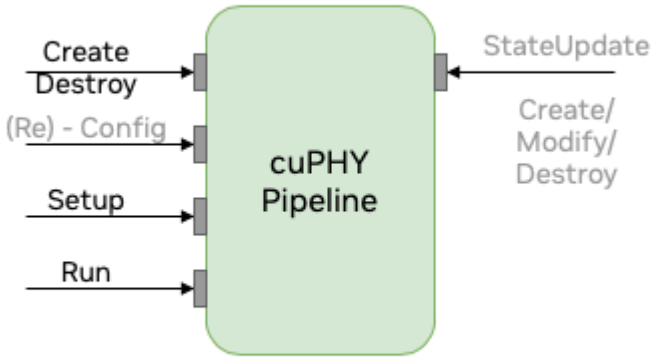


cuPHY library within 5G NR software stack

cuPHY is optimized to take advantage of the massive parallel processing capability of the GPU architecture by running the workloads in parallel when possible. cuPHY driver orchestrates signal processing tasks running on the GPU. These tasks are organized according to the PHY layer channel type, e.g. PDSCH, PUSCH, SSB, etc. A task related to a given channel is termed as pipeline. For example, PDSCH channel is processed in PDSCH

pipeline and the PUSCH channel is processed in PUSCH pipeline. Each pipeline includes a series of functions related to the specific pipeline and consists of multiple CUDA kernels. Each pipeline is capable of running signal processing workloads for multiple cells. The pipelines are dynamically managed for each slot by cuPHY driver with channel aggregate objects. The group of cuPHY channel pipelines that is executed in a given time slot depends on what is scheduled by the L2 in that time slot.

The cuPHY library exposes a set of APIs per PHY channel to create, destroy, setup, configure and run each pipeline as shown in the following figure. L2 adapter translates SCF FAPI messages and other system configurations and cuPHY driver invokes associated cuPHY APIs for each slot. The API's shown as grey such as (Re)-Config, StateUpdate are not currently supported.



cuPHY API interface

The following are descriptions of the APIs in the above figure:

- *Create:* performs pipeline construction time operations, such as PHY and CUDA object instantiation, memory allocations, etc.
- *Destroy:* executes teardown procedures of a pipeline and frees allocated resources.
- *Setup:* sets up PHY descriptors with slot information and batching

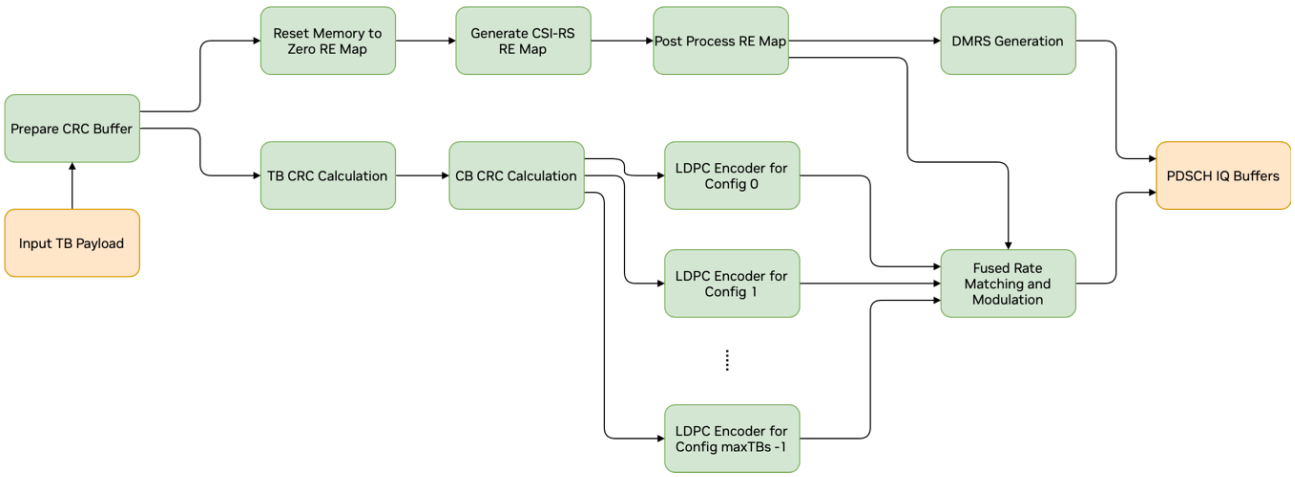
needed to execute the pipeline.

- *Run:* launches a pipeline.

The following sections provide more details on the implementation of each cuPHY channel pipeline.

PDSCH Pipeline

The PDSCH pipeline receives configuration parameters for each cell and the UE and the corresponding DL transport blocks (TBs). After completing the encoding of the PDSCH channel, the pipeline outputs IQ samples mapped to the resource elements (REs) allocated to the PDSCH. The PDSCH pipeline consists of multiple CUDA kernels, which are launched with CUDA graph functionality to reduce the kernel launch overhead. The diagram of the CUDA graph used by PDSCH pipeline is shown in the following figure. The green boxes represent CUDA kernels and the orange boxes represent input and output buffers.



Graph Diagram of the PDSCH Pipeline

The PDSCH pipeline contains the following components:

- CRC calculation of the TBs and code-blocks (CBs)
- LDPC encoding
- Fused Rate Matching and Modulation Mapper
- DMRS generation

The CRC calculation component performs the code block segmentation and the CRC calculation. The CRC is calculated first for each TB and then for each CB. The fused rate matching and modulation component performs rate-matching, scrambling, layer-mapping, pre-coding and modulation. This component is also aware of which resource elements it should skip if CSI-RS is configured.

The PDSCH pipeline involves the following kernels:

- prepare_crc_buffers
- crcDownlinkPdschTransportBlockKernel
- crcDownlinkPdschCodeBlocksKernel
- ldpc_encode_in_bit_kernel
- fused_dl_rm_and_modulation
- fused_dmrs

Kernels exercised only if CSI-RS parameters are present are as follows:

- zero_memset_kernel
- genCsirsReMap
- postProcessCsirsReMap

The cuPHY PDSCH transmit pipeline populates parts of a 3D tensor buffer of I/Q samples in GPU memory, where each sample is a complex number using fp16, i.e. each sample is a __half2 using x for the real part and y for the imaginary part. The output 3D tensor buffer is allocated by the cuPHY driver when the application is first launched and it is reset for every slot (i.e., between successive PDSCH launches) by the cuPHY driver. Here, re-setting the buffer means, it is initialized to all zero values.

The output tensor contains 14 symbols on time domain (x-axis), 273 PRBs (Physical Resource Blocks) on frequency domain (y-axis), and up to 16 layers on spatial domain (z-axis). For the y-axis, each PRB contains 12 REs, and each RE is a __half2 data. Contiguous PRBs for the same OFDM symbol and spatial layer are allocated next to each other on memory. The resources are mapped in memory in the following order: frequency domain, time domain and then the spatial domain (or layer domain). This is the maximum size of the output buffer needed for a cell per slot.

The PDSCH only fills in parts of that buffer, i.e., its allocated PRBs, based on various configuration parameters it receives that vary over time. Parts of the slot can be filled by other downlink control channels. From a PDSCH standpoint, only the two fused_* kernels listed above, fused_dl_rm_and_modulation and fused_dmrs write to the output buffer. The fused rate-matching and modulation kernel writes data part of the I/Q samples, while the DMRS kernel only writes the DMRS symbols, i.e., only 1 or 2 contiguous symbols in the x-dimension. Note that, unlike other components, DMRS is not dependent on any of the previous pipeline stages.

The PDSCH pipeline expects pre-populated structs cuphyPdschStatPrms_t (cuPHY PDSCH static parameters) and cuphyPdschDynPrms_t (cuPHY PDSCH dynamic parameters) that include the input data and the necessary configuration parameters.

The TB data input can exist either in CPU or GPU memory depending on the cuphyPdschDataIn_t.pBufferType. If this is GPU_BUFFER, then the host to device (H2D) memory copies for that data can happen before PDSCH setup is executed for each cell. This is called prepone H2D copy and it can be configured by setting the prepone_h2d_copy flag in the l2_adapter_config_*.yaml file. If prepone H2D copy is not enabled, the copy operations happen as part of PDSCH setup. It is highly recommended that the prepone H2D copy should be enabled to achieve high capacity in a multiple cell scenario.

The way LDPC kernels are initiated can change when multiple TBs are configured on PDSCH. If the LDPC configuration parameters are identical across TBs, PDSCH launches a single LDPC kernel for all TBs (as it is the case for the other PDSCH components). If the LDPC configuration parameters vary across the TBs, then multiple LDPC kernels are launched, one for each unique configuration parameters set. Each LDPC kernel is launched on a separate CUDA stream.

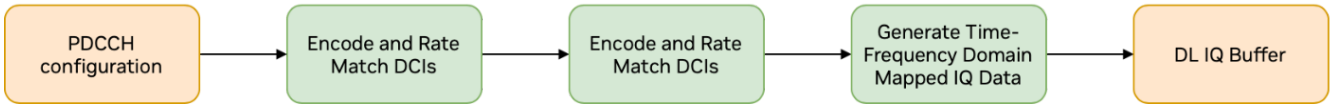
The PDSCH CUDA graph contains only kernel nodes and has the layout shown in the PDSCH graph diagram shown above. As it is not possible to dynamically change the graph geometry at runtime, PDSCH_MAX_HET_LDPC_CONFIGS_SUPPORTED potential LDPC kernel nodes are created. Depending on the LDPC configuration parameters and the number of TBs, only a subset of these kernels perform LDPC encoding. The remaining nodes are disabled at runtime if needed per PDSCH. The DMRS kernel node is not dependent on any of the other PDSCH kernels. Therefore, it can be placed anywhere in the graph. The three kernels preceding the DMRS in the graph are only exercised if CSI-RS parameters are present (or CSI-RS is configured). These kernels compute information needed by the fused rate matching and modulation kernel about the REs that need to be skipped.

PDCCH Pipeline

The cuPHY PDCCH channel processing involves the following kernels:

- encodeRateMatchMultipleDCIsKernel
- genScramblingSeqKernel
- genPdcchTfSignalKernel

When running in graphs mode, the CUDA graph launched on every slot contains only kernel nodes and its current layout is as depicted in the below figure.



cuPHY PDCCH graph layout

PDCCH kernel takes static and dynamic parameters as in PDSCH.

Notes on PDCCH configuration and dataset conventions:

- The PdcchParams dataset contains the coreset parameters for a given cell. Dataset DciParams_coreset_0_dci_0 contains the DCI parameters for the first DCI of coreset 0. There is a separate dataset for every DCI in a cell with the naming convention: DciParams_coreset_<i>_dci_<j>, where i has values from 0 up to (number of coresets – 1), while j starts from 0 for every coreset i and goes up to (PdcchParams[i].numDlDci – 1) for that coreset.
- Dataset DciPayload_coreset_0_dci_0 contains the DCI payload, in bytes, for the first DCI of coreset 0. It follows the naming convention mentioned above DciParams_coreset_0_dci_0.
- Dataset(s) DciPmW_coreset_i_dci_j hold the precoding matrix for a given DCI, coreset pair, if it has precoding enabled.
- X_tf_fp16 is the 3D output tensor for that cell and is used for reference checks in the various PDCCH examples.
- X_tf_cSamples_bfp* datasets that contain compressed data are not used in cuPHY, since compression happens in cuphydriver after all cuPHY processing for all

downlink channels scheduled in a slot has completed.

SSB Pipeline

The cuPHY SS Block channel processing involves the following kernels:

- encodeRateMatchMultipleSSBsKernel
- ssbModTfSigKernel

When running in graphs mode, the CUDA graph launched on every slot contains only these two kernel nodes connected in sequence.

Notes on SSB configuration and dataset conventions:

- The SSTxParams dataset contains all the nSsb, SSB parameters for a given cell.
- SSB bursts cannot be multiplexed in frequency domain, they can only be multiplexed in time domain.
- nSsb datasets contains the number of SSBs in a cell, this is also the size of the SSTxParams dataset.
- x_mib contains the Master Information Block (MIB) for each SSB in the cell as an uint32_t element; only the least significant 24-bits of each element are valid.
- Dataset(s) Ssb_PM_W* contain the precoding matrices if precoding is enabled for a given SSB.
- X_tf_fp16 is the 3D output tensor for that cell and is used for reference checks in the various SSB examples. Every I/Q sample there is stored as __half2c.
  X_tf is similar to X_tf_fp16 but every I/Q sample there is stored as float2 instead of __half2; not currently used in cuPHY.
- X_tf_cSamples_bfp* datasets hold the output compressed and are not used in cuPHY as compression is applied as part of the cuphydriver.

CSI-RS Pipeline

The cuPHY CSI-RS channel processing involves the following kernels:

- genScramblingKernel
- genCsirsTfSignalKernel

When running in graphs mode, the CUDA graph launched on every slot contains only these two kernel nodes connected in sequence.

Notes on CSI-RS configuration and dataset conventions:

- CsirsParamsList contains configuration parameters which are used for non-zero power signal generation (e.g., NZP, TRS).
- Please note that CsirsParamsList dataset can have multiple elements. All elements in the dataset can be processed with single setup/run call.
- X_tf_fp16 is the 3D reference output tensor for that cell and is used for reference checks in the various CSI-RS examples. Every I/Q sample there is stored as __half2c.
- X_tf is similar to X_tf_fp16 but every I/Q sample there is stored as float2 instead of __half2; not currently used in cuPHY.
- X_tf_cSamples_bfp* datasets hold the output compressed and are not used in cuPHY as compression is applied as part of cuphydriver.
- X_tf_remap is reference output for RE Map, this is not used currently as current implementation only generates NZP signal.
- Dataset(s) Csirs_PM_W* contain precoding matrices and are used if precoding is enabled.

PUSCH Pipeline

The PUSCH pipeline includes the following components (which are illustrated in the *PUSCH Pipeline Front End <figure_pusch_pipeline_front_end>* and *PUSCH and CSI Part 1 Decoding <figure_pusch_csi_part_1_decoding>* figures):

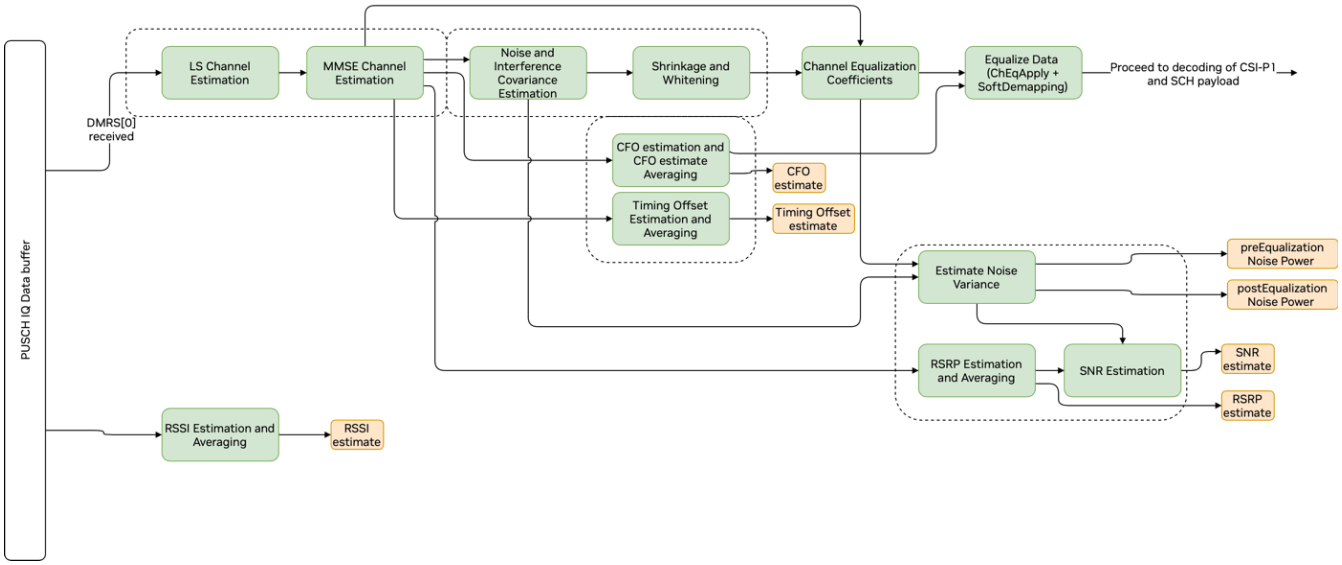
- Least squares (LS) channel estimation
- Minimum Mean Square Error (MMSE) channel estimation
- Noise and interference covariance estimation

- Shrinkage and whitening
- Channel Equalization
- Carrier frequency offset (CFO) estimation and CFO averaging
- Timing offset (TO) estimation and averaging.
- Received signal strength indicator (RSSI) estimation and averaging
- Noise variance estimation
- Received signal received power (RSRP) estimation and averaging
- SNR estimation
- De-rate matching
- LDPC backend

If CSI part 2 is configured, the following components are also used (these components are illustrated in the *PUSCH and CSI Part 1 Decoding <figure_pusch_csi_part_1_decoding>* and *PUSCH and CSI Part 2 Decoding <figure_pusch_csi_part_2_decoding>* figures):

- Simplex decoder or RM decoder or Polar decoder (for CSI decoding of CSI part 1 depending on the UCI payload size)
- CSI part 2 de-scrambling and de-rate matching
- Simplex decoder or RM decoder or Polar decoder (for CSI decoding of CSI part 2 depending on the UCI payload size)

The PUSCH pipeline receives IQ samples, which are provided by order and decompression kernels. The received IQ data is stored in the address cuphyPuschDataIn_t PhyPuschAggr::DataIn.pTDataRx as the cuphyTensorPrm_t type. The IQ samples are represented by half precision (16-bits) real and imaginary values. The size of the input buffer is multiplication of number of maximum PRBs (273), number of subcarriers per PRB (12), number of OFDM symbols per slot (14) and number of maximum antenna ports per cell (16). This buffer is created for each cell.



Graph Diagram of the PUSCH Pipeline Front End

Channel Estimation

First Stage (LS CE)	
Input Buffer	PhyPuschAggr::DataIn.pTDataRx
Data type	CUPHY_C_16_F : tensor vector of IQ samples
Dimensions	[(ORAN_MAX_PRB*CUPHY_N_TONES_PER_PRB), OFDM_SYMBOLS_PER_SLOT, MAX_AP_PER_SLOT]: [(273*12),14,16]
Description	IQ samples of the input data received from the FH for an UL slot. The I/Q data are represented in half precision float.
Output Buffer	PuschRx::m_tRefDmrsLSEstVec[i] *Note:* the index i refers to a PRB range (or UE group)
Data type	CUPHY_C_32_F: float complex IQ samples
Dimensions	[(CUPHY_N_TONES_PER_PRB*(number_of PRBs)/2), NUM_LAYERS, NUM_ANTENNAS, NH]: [(12*(number of PRBs)/2), (number of layers), (number of RX antennas), (number of DMRS symbols)]
Description	IQ samples of the initial channel estimates on DMRS symbols. The I/Q data are represented in half precision float.
Output Buffer	PuschRx::m_tRefDmrsAccumVec[i] *Note:* the index i refers to a PRB range (or UE group)

Data Type	CUPHY_C_32_F: float complex IQ samples
Dimensions	[1,2]: Two dimensions for one active and one non-active buffer
Description	Holds summation of conj(H_ls[k])*H_ls[k+1] in a given PRB range, which is then used to calculate mean delay in the next stage. The index k refers to the subcarrier index in a given PRB range. conj() represents the conjugation function.

Channel estimation (CE) consists of two stages: least-squares (LS) CE and minimum-mean-square (MMSE) CE.

In the LS CE stage, DMRS symbols are used to obtain initial channel estimate on DMRS REs and to calculate mean delay of the channel impulse response (CIR). The mean delay and the initial estimates are then used to obtain channel estimates in data REs on the second stage with MMSE filtering operation.

The second stage invokes a dispatch kernel chEstFilterNoDftSOfdmDispatchKernel() to support different configurations. The dispatch kernel first calculates mean channel delay by using the stored value tInfoDmrsAccum from the first stage. It then chooses an appropriate kernel depending on number of PRBs in the given PUSCH allocation and number of consecutive DMRS symbols (drvdUeGrpPrms.dmrsMaxLen). The MMSE filtering operation is done by the kernel windowedChEstFilterNoDftSOfdmKernel().

Second Stage (MMSE CE)	
Input Buffer	Receives outputs of the second stage as input.
Input Buffer	statDescr.tPr mFreqInterpCoefsSmall, statDescr.tPrmFreqInterpCoefs, or statDescr.tPrmFreqInterpCoefs4
Description	Interpolation filter coefficients depending on the number of PRBs
Output Buffer	PuschRx::m_tRefHEstVec[i] *Note:* the index i refers to a PRB range (or UE group)
Data type	CUPHY_C_32_F: float complex IQ samples
Dimensions	[NUM_ANTENNAS, NUM_LAYERS, NF, NH]: [(number of RX antennas), (number of layers), (12*(number of PRBs)), (number of DMRS symbols)]
Description	Estimates of the received channel on the DMRS symbols.

Noise and Interference Covariance Estimation

Input Buffer	Receives outputs of channel estimation kernel as input.
Output Buffer	PuschRx:: m_tRefNoiseVarPreEq
Data type	CUPHY_R_32_F: float real values
Dimensions	[1, NUM_UE_GROUPS]
Description	Estimates of the noise variance pre-equalization per UE group (or PRB range).
Output Buffer	PuschRx:: m_tRefLwInvVec[i] *Note:* the index i refers to a PRB range (or UE group)
Data Type	CUPHY_C_32_F: float complex IQ samples
Dimensions	[NUM_ANTENNAS, NUM_ANTENNAS, numPRB]: [(number of RX antennas), (number of RX antennas),(number of PRBs)]
Description	Inverse Cholesky factor of noise-interference tensor information.

Carrier Frequency and Timing Offset Estimation

Input Buffers	PuschRx::m_tRefHEstVec[i] This buffer is received from Channel Estimation kernel. *Note:* the index i refers to a PRB range (or UE group).
Output Buffer	PuschRx:: m_tRefCfoEstVec[i] *Note:* the index i refers to a PRB range (or UE group)
Data Type	CUPHY_R_32_F: float real values
Dimensions	[MAX_ND_SUPPORTED, (number of UEs)]: [14, (number of UEs)]
Description	CFO estimate vector.
Output Buffer	PuschRx:: m_tRefCfoHz
Data Type	CUPHY_R_32_F: float real values.
Dimensions	[1, (number of UEs)]
Descriptions	CFO estimate values in Hz.

Output Buffer	PuschRx:: m_tRefTaEst
Data Type	CUPHY_R_32_F: float real values.
Dimensions	[1, (number of UEs)]
Descriptions	Timing offset estimates.
Output Buffer	PuschRx:: m_tRefCfoPhaseRot
Data Type	CUPHY_C_32_F: float complex values.
Dimensions	[CUPHY_PUSCH_RX_MAX_N_TIME_CH_EST, CUPHY_PUSCH_RX_MAX_N_LAYERS_PER_UE_GROUP, MAX_N_USER_GROUPS_SUPPORTED] : [(max number of channel estimates in time, =4), (max layers per UE group, =8), (max UE groups, =128)]
Descriptions	Carrier offset phase rotation values
Output Buffer	PuschRx:: m_tRefTaPhaseRot
Data Type	CUPHY_C_32_F: float complex values.
Dimensions	[1, CUPHY_PUSCH_RX_MAX_N_LAYERS_PER_UE_GROUP] : [1, (max layers per UE group, =8)]
Descriptions	Carrier offset phase rotation values

Soft De-mapper

Channel Equalization Coefficients Computation Kernel	
Input Buffers	PuschRx::m_tRefHEstVec[i], PuschRx:: m_tRefLwInvVec[i], PuschRx:: m_tRefCfoEstVec[i] These buffers are received from Noise and Interference Covariance Estimation, Channel Estimation and CFO Estimation kernels. *Note:* the index i refers to a PRB range (or UE group).
Output Buffer	PuschRx:: m_tRefReeDiagInvVec[i] *Note:* the index i refers to a PRB range (or UE group)

Data Type	CUPHY_R_32_F: float real values
Dimensions	[CUPHY_N_TONES_PER_PRB, NUM_LAYERS, NUM_PRBS, nTimeChEq]: [12*(number of PRBs), (number of layers), (number of PRBs), (number of time domain estimates)]
Description	Channel equalizer residual error vector.
Output Buffer	PuschRx:: m_tRefCoefVec[i] *Note:* the index i refers to a PRB range (or UE group)
Data Type	CUPHY_C_32_F: float complex IQ samples
Dimensions	[NUM_ANTENNAS, CUPHY_N_TONES_PER_PRB, NUM_LAYERS, NUM_PRBS, NH]: [(number of RX antennas), 12*(number of PRBs), (number of layers), (number of PRBs), (number of DMRS positions)]
Descriptions	Channel equalizer coefficients.

Channel Equalization MMSE Soft De-mapping Kernel	
Input Buffers	PuschRx:: m_tRefCoefVec[i], PuschRx:: m_tRefCfoEstVec[i], PuschRx:: m_tRefReeDiagInvVec[i] PuschRx:: m_drvdUeGrpPrmsCpu[i].tInfoDataRx These buffers are received from Noise and Interference Covariance Estimation, Channel Estimation and CFO Estimation kernels. *Note:* the index i refers to a PRB range (or UE group).
Output Buffer	PuschRx:: m_tRefDataEqVec[i] *Note:* the index i refers to a PRB range (or UE group)
Data Type	CUPHY_C_16_F : tensor vector of half float IQ samples.
Dimensions	[NUM_LAYERS, NF, NUM_DATA_SYMS]: [(number of layers), 12* (number of PRBs), (number of data OFDM symbols)]
Description	Equalized QAM data symbols.
Output Buffer	PuschRx:: m_tRefLLRVec[i] *Note:* the index i refers to a PRB range (or UE group)

Data Type	CUPHY_R_16_F : tensor vector of half float real samples.
Dimensions	[CUPHY_QAM_256, NUM_LAYERS, NF, NUM_DATA_SYMBOLS]: [(number of bits for 256QAM = 8), (number of layers), (number of layers), 12*(number of PRBs), (number of data OFDM symbols)]
D escriptions	Output LLRs or softbits. Used if UCI on PUSCH is enabled.
Output Buffer	PuschRx:: m_tRefLLRCdm1Vec[i] *Note:* the refers to a PRB range (or UE group)index i
Data Type	CUPHY_R_16_F : tensor vector of half float real samples.
Dimensions	[CUPHY_QAM_256, NUM_LAYERS, NF, NUM_DATA_SYMBOLS]: [(number of bits for 256QAM = 8), (number of layers), (number of layers), 12*(number of PRBs), (number of data OFDM symbols)]
D escriptions	Output LLRs or softbits. Used if there is no UCI on PUSCH.

De-rate matching and Descrambling

Input Buffer	PuschRx:: m_tRefLLRVec[i] or PuschRx:: m_tRefLLRCdm1Vec[i], PuschRx:: m_pTbPrmsGpu
Output Buffer	PuschRx:: m_pHarqBuffers
Data type	uint8_t
Dimensions	Function of TB size and number of TBs.
Description	Rate-matching/descrambling output. It is on a host pinned GPU memory. It is mapped to PhyPuschAggr::DataInOut.pHarqBuffersInOut

RSSI Estimation

The RSSI is calculated from the received signal by first calculating the received signal power on each RE and each receive antenna. The total power is then calculated by summation of powers across the frequency resources and receive antennas and average over OFDM symbols in accordance to the SCF FAPI specification.

Input Buffer	PuschRx:: m_drvdUeGrpPrmsCpu[i].tInfoDataRx

Output Buffer	PuschRx:: m_tRefRssiFull
Data type	CUPHY_R_32_F : tensor vector of float real samples.
Dimensions	[MAX_ND_SUPPORTED, MAX_N_ANTENNAS_SUPPORTED , nUEgroups]: [(max number of time domain estimates, =14), (max number of antennas, =64), (number of UE groups)]
Description	Measured RSSI (per symbol, per antenna, per UE group).
Output Buffer	PuschRx:: m_tRefRssi
Data type	CUPHY_R_32_F : tensor vector of float real samples.
Dimensions	[1, nUEgroups]:[1, (number of UE groups)]
Description	Measured RSSI per UE group.

RSRP and SINR Estimation

Input Buffer	PuschRx::m_tRefHEstVec[i], PuschRx:: m_tRefReeDiagInvVec[i], PuschRx:: m_tRefNoiseVarPreEq
Output Buffer	PuschRx:: m_tRefRsrp
Data type	CUPHY_R_32_F : tensor vector of float real samples.
Dimensions	[1, nUEgroups]:[1, (number of UE groups)]
Description	RSRP values across UEs.
Output Buffer	PuschRx:: m_tRefNoiseVarPostEq
Data type	CUPHY_R_32_F : tensor vector of float real samples.
Dimensions	[1, nUEgroups]:[1, (number of UE groups)]
Description	Post-equalization noise variances across UEs
Output Buffer	PuschRx:: m_tRefSinrPreEq
Data type	CUPHY_R_32_F : tensor vector of float real samples.
Dimensions	[1, nUEgroups]:[1, (number of UE groups)]
Description	Pre-equalization SINR values across UEs.

Output Buffer	PuschRx:: m_tRefSinrPostEq
Data type	CUPHY_R_32_F : tensor vector of float real samples.
Dimensions	[1, nUEgroups]:[1, (number of UE groups)]
Description	Post-equalization SINR values across UEs.

UCI on PUSCH Decoder

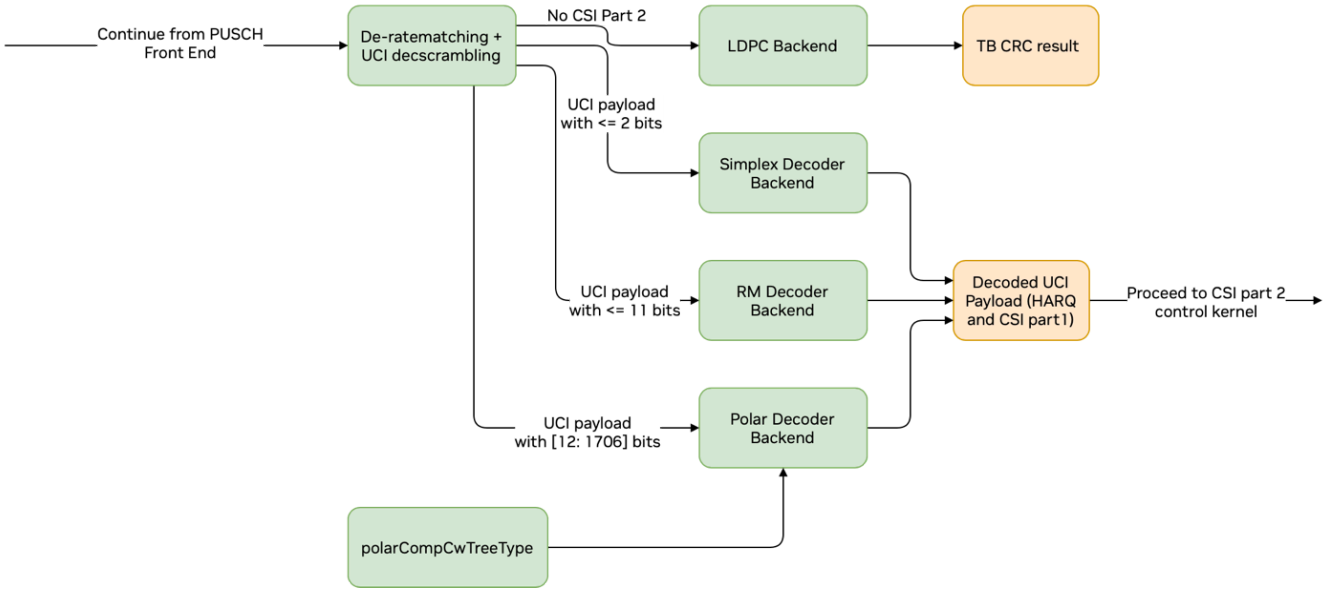
If UCI is configured on PUSCH channel, output of the soft-demapper first goes through de-segmentation to separate HARQ, CSI part 1 and CSI part 2 and SCH softbits (or LLRs). This initial step is done by the kernel uciOnPuschSegLLRs0Kernel().

If CSI-part2 is present, CSI-part2 control kernel is launched as shown in the figure below as a dashed box. This kernel determines the number of CSI-part2 bits and rate-matched bits and selects the correct decoder kernels and initiates their setup functions.

De-segmentation of CSI-part2 payload is done by uciOnPuschSegLLRs2Kernel() kernel, which separates CSI-part2 UCI and SCH softbits.

UCI on PUSCH De-segmentation of First Phase	
Input Buffer	PuschRx:: m_tPrmLLRVec[i]
Output Buffer	PuschRx::m_pTbPrmsGpu->pUePrmsGpu[i].d_harqLLrs;
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	HARQ soft bits.
Output Buffer	PuschRx::m_pTbPrmsGpu->pUePrmsGpu[ueIdx].d_csi1LLRs;
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	CSI part 1 soft bits.
Output Buffer	PuschRx::m_pTbPrmsGpu->pUePrmsGpu[i]. d_schAndCsi2LLRs

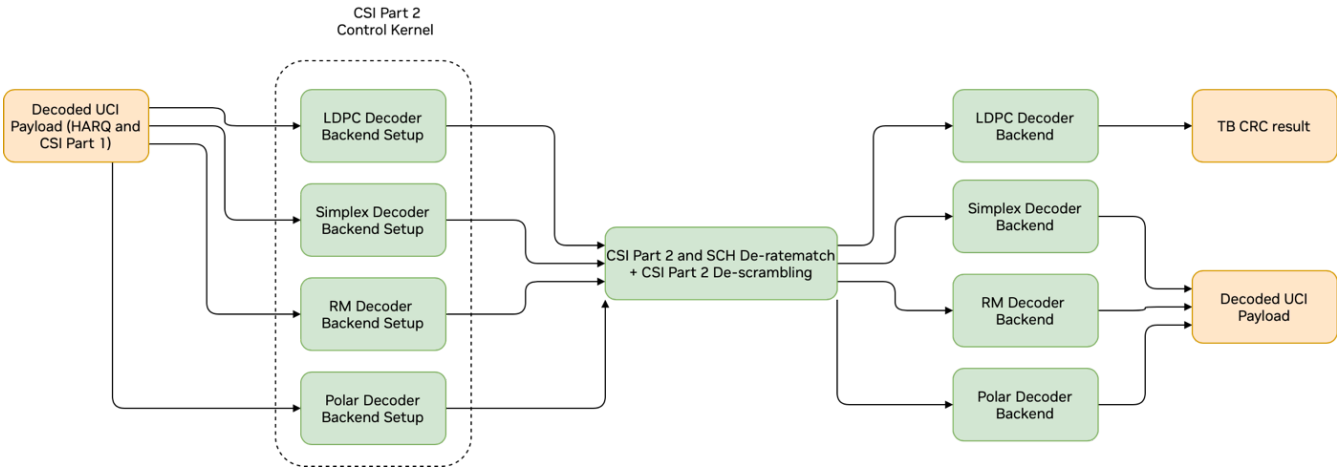
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Shared channel (SCH) and CSI part 2 soft bits.



Graph Diagram of the PUSCH and CSI Part 1 Decoding

UCI on PUSCH De-segmentation of Second Phase	
Input Buffer	PuschRx:: m_tPrmLLRVec[i]
Output Buffer	P uschRx::m_pTbPrmsGpu->pUePrmsGpu[i].d_schAndCsi2LLRs;
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Pointer to SCH softbits
Output Buffer	PuschRx::m_pTbPrmsGpu->pUePrmsGpu[i].d_schAndCsi2LLRs + PuschRx::m_pTbPrmsGpu->pUePrmsGpu[i].G;
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.

Description	Pointer to CSI part2 softbits



Graph Diagram of the PUSCH and CSI Part 2 Decoding

Simplex Decoder

The simplex decoder implements maximum likelihood (ML) decoder. It receives input LLRs and outputs estimated codewords. It also reports HARQ DTX status.

Input Buffer	PuschRx:: m_pSpxCwPrmsCpu[spxCwIdx].d_LLRs
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Pointer to input LLRs
Output Buffer	PuschRx:: m_pSpxCwPrmsCpu[spxCwIdx].d_cbEst
Data type	uint32_t*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Decoded UCI payload.
Output Buffer	PuschRx:: m_pSpxCwPrmsCpu[spxCwIdx].d_DTXStatus
Data type	Uint8_t*
Dimensions	Parameter.
Description	Pointer to HARQ detection status.

Reed Muller (RM) Decoder

The RM decoder implements maximum likelihood (ML) decoder. It receives input LLRs and outputs estimated codewords. It also reports HARQ DTX status.

Input Buffer	PuschRx:: m_pSpxCwPrmsCpu[rmCwIdx].d_LLRs
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Pointer to input LLRs
Output Buffer	PuschRx:: m_pSpxCwPrmsCpu[rmCwIdx].d_cbEst
Data type	uint32_t*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Decoded UCI payload.
Output Buffer	PuschRx:: m_pSpxCwPrmsCpu[rmCwIdx].d_DTXStatus
Data type	Uint8_t*
Dimensions	Parameter.
Description	Pointer to HARQ detection status.

Polar Decoder

Polar decoder uses CRC aided list decoder with tree pruning. There are many variants of the decoding algorithm that is used in decoding of Polar codes. Please see [2, 3] for some of the related work. The exact implementation in cuPHY is optimized for the GPU architecture.

The tree-pruning algorithms combine leaf nodes together, which is a better data structure for execute decoding in parallel. Hence it is more suitable for GPU architecture. There are different methods of forming leaf nodes in the tree pruning algorithm. In our implementation we use rate-0 and rate-1 leaf codewords. In rate-0 leaf nodes, multiple bits are always frozen and are zero, whereas there are no frozen bits in rate-1 leaf nodes. In rate-1 codewords, LLRs can be decoded in parallel.

Tree pruning is done by compCwTreeTypesKernel()before the input LLRs are received by the Polar Decoder kernel.

If the list size is equal to 1, polarDecoderKernel(), if the list size is greater than 1, listPolarDecoderKernel()is run.

Input Buffer	PuschRx:: m_cwTreeLLRsAddrVec
Data type	__half*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Pointer to codeword tree of LLR addresses.
Output Buffer	PuschRx:: m_cbEstAddrVec
Data type	uint32_t*
Dimensions	Single dimensional array, the size depending on the payload.
Description	Pointer to estimated CB addresses.

LDPC Decoder

LDPC decoder is implemented with normalized layered min-sum algorithm [1] and it uses short float (FP16) data type as log-likehood ratio (LLR) metrics.

Input Buffer	PuschRx:: m_LDPCDecodeDescSet.llr_input[m_LDPCDecodeDescSet .num_tbs] The first address is also mapped to PuschRx::m_pHarqBuffers[ueIdx]
Data type	cuphyTransportBlockLLRDesc_t
Dimensions	Single dimensional array, the size depending on the number of valid TB descriptors. The max size is 32.
Description	Input LLR buffers.
Output Buffer	PuschRx:: m_LDPCDecodeDescSet.tb_output[m_LDPCDecodeDescSet .num_tbs] The first address is also mapped to PuschRx::d_LDPCOut + offset Offset is a function of UE index and number of codewords per UE.
Data type	cuphyTransportBlockDataDesc_t
Dimensions	Single dimensional array, the size depending on the number of valid TB descriptors.
Description	Pointer to estimated TB addresses.

CRC Decoder

Code Block CRC Decoder Kernel	
Input Buffer	PuschRx::d_pLDPCOut, PuschRx:: m_pTbPrmsGpu
Descriptions	LDPC decoder output and TB parameters needed to decode the CRC.
Output Buffer	PuschRx:: m_outputPrms.pCbCrcsDevice;
Data type	uint32_t
Dimensions	[1, total number of CBs (across UEs)]
Description	CRC output.
Output Buffer	PuschRx:: m_outputPrms.pTbPayloadsDevice
Data type	Uint8_t
Dimensions	[1, total number of TB payload bytes]
Description	TB payload.
Transport Block CRC Decoder Kernel	
Input Buffer	PuschRx:: m_outputPrms.pTbPayloadsDevice, PuschRx:: m_pTbPrmsGpu
Output Buffer	PuschRx:: m_outputPrms.pTbCrcsDevice
Data Type	uint32_t
Dimensions	[1, total number of TBs (across UEs)]
Description	TB CRC output.

PUCCH Pipeline

The PUCCH pipeline can be divided into logical stages. The first, front-end processing, is unique for each PUCCH format and involves descrambling and demodulation to recover transmitted symbols. For formats 0 and 1, this is the only stage performed as there is no decoding necessary to recover data. For formats 2 and 3, this is followed by decoding.

Here, the kernels used are the same as those in PUSCH for the same decoding type. Finally, the decoded data is segmented into HARQ, SR and CSI payloads.

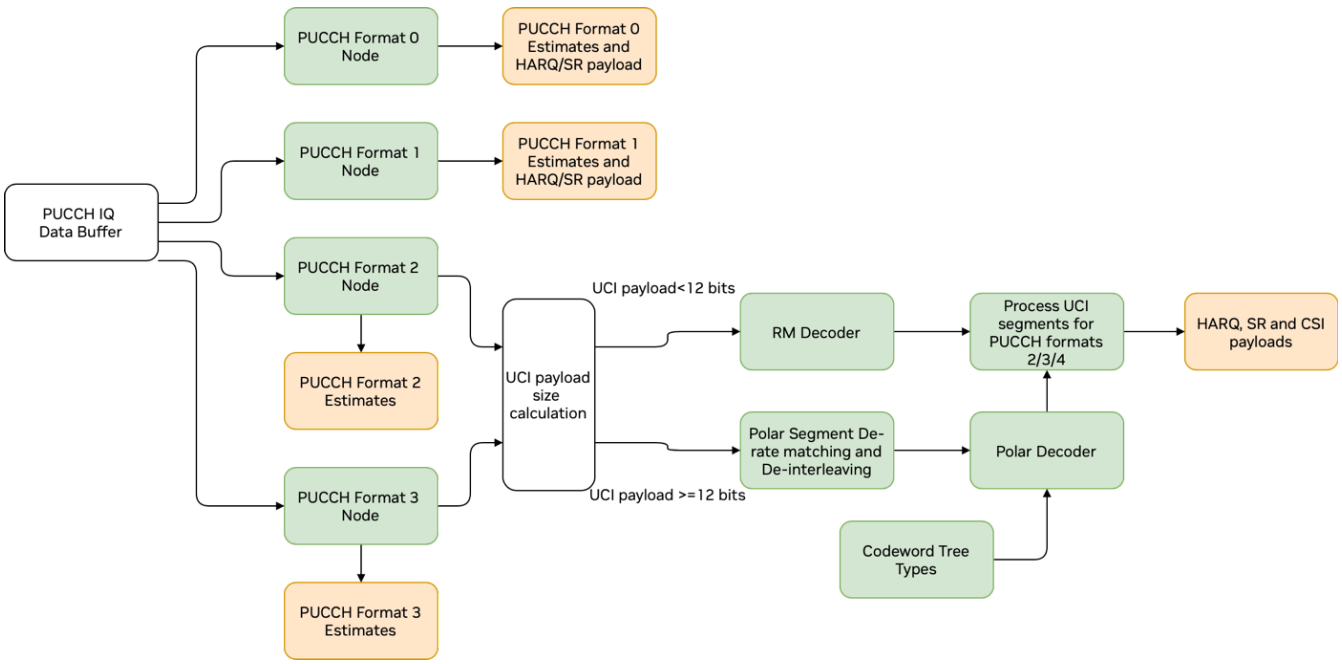
The kernels responsible for front-end processing are as follows:

- pucchF0RxKernel
- pucchF1RxKernel
- pucchF2RxKernel
- pucchF3RxKernel

with each corresponding to formats 0 through 3 respectively. For formats 0 and 1, hard decisions are made as part of demodulation to recover 1 or 2 payload bits, depending on specific configuration. For formats 2 and 3, LLRs are recovered from demodulation and used for decoding. Each front-end processing kernel also calculates RSSI, and RSRP and uses DMRS to perform SINR, interference, and timing advance estimation.

For formats 2 and 3, payloads less than 12 bits in length are handled by the Reed Muller decoder kernel detailed in Section 3.6.10. Payloads of 12 bits and larger are handled by a de-rate matching and de-interleaving kernel (polSegDeRmDeItlKernel) and then processed by the polar decoder kernel detailed in Section 3.6.11.

Finally, formats 2 and 3 decoded payloads are segmented by a segmentation kernel (pucchF234UciSegKernel) to recover the corresponding HARQ, SR, and CSI payloads.



Graph Diagram of the PUCCH Pipeline

Input Buffer	PucchRx::m_tPrmDataRxBufCpu[i].tInfoDataRx
Data type	CUPHY_C_16_F : tensor vector of IQ samples
Dimensions	[(ORAN_MAX_PRB*CUPHY_N_TONES_PER_PRB), OFDM_SYMBOLS_PER_SLOT, MAX_AP_PER_SLOT]
Output Buffer	PucchRx::m_outputPrms.pF0UciOutGpu
Data type	cuphyPucchF0F1UciOut_t*
Dimensions	Single dimensional array of length equal to the number of format 0 UCIs
Description	HARQ values and estimator measurements, including SINR, Interference, RSSI, RSRP (in dB) and timing advance (in uSec) per UCI
Output Buffer	PucchRx::m_outputPrms.pF0UciOutGpu
Data type	cuphyPucchF0F1UciOut_t*
Dimensions	Single dimensional array of length equal to the number of format 1 UCIs
Description	HARQ values and estimator measurements, including SINR, Interference, RSSI, RSRP (in dB) and timing advance (in uSec) per UCI
Output Buffer	PucchRx:: m_tSinr
Data type	CUPHY_R_32_F : tensor vector of float values.
Dimensions	[(number of format 2 & 3 UCIs)]
Description	Measured SINR per UCI (in dB)
Output Buffer	PucchRx:: m_tRssi
Data type	CUPHY_R_32_F : tensor vector of float values.
Dimensions	[(number of format 2 & 3 UCIs)]
Description	Measured RSSI per UCI (in dB)
Output Buffer	PucchRx:: m_tRsrp
Data type	CUPHY_R_32_F : tensor vector of float values.

Dimensions	[(number of format 2 & 3 UCIs)]
Description	Measured RSRP per UCI (in dB)
Output Buffer	PucchRx:: m_tInterf
Data type	CUPHY_R_32_F : tensor vector of float values.
Dimensions	[(number of format 2 & 3 UCIs)]
Description	Measured Interference per UCI (in dB)
Output Buffer	PucchRx:: m_tNoiseVar
Data type	CUPHY_R_32_F : tensor vector of float values.
Dimensions	[(number of format 2 & 3 UCIs)]
Description	Measured Noise Variance per UCI (in dB)
Output Buffer	PucchRx:: m_tTaEst
Data type	CUPHY_R_32_F : tensor vector of float values.
Dimensions	[(number of format 2 & 3 UCIs)]
Description	Measured Timing Advance per UCI (in uSec)
Output Buffer	PucchRx::m_tUciPayload
Data type	CUPHY_R_8U : tensor vector of unsigned bytes
Dimensions	[(total number payload bytes for format 2 & 3 UCIs rounded up to 4-byte words for each payload)]
Description	Format 2 & 3 UCI payloads rounded to 4-byte words. If 1 UCI has HARQ & CSI-P1 of 1 bit each, they will each get a 4-byte word for a total of 8 bytes.
Output Buffer	PucchRx:: m_tHarqDetectionStatus
Data type	CUPHY_R_8U : tensor vector of unsigned bytes
Dimensions	[(number of format 2 & 3 UCIs)]
Description	HARQ detection status
Output Buffer	PucchRx:: m_tCsiP1DetectionStatus

Data type	CUPHY_R_8U : tensor vector of unsigned bytes
Dimensions	[(number of format 2 & 3 UCIs)]
Description	CSI Part 1 detection status
Output Buffer	PucchRx:: m_tCsiP2DetectionStatus
Data type	CUPHY_R_8U : tensor vector of unsigned bytes
Dimensions	[(number of format 2 & 3 UCIs)]
Description	CSI Part 2 detection status

PRACH Pipeline

The PRACH pipeline uses IQ samples segmented for each occasion and performs detection and estimation for configured PRACH signals. This process operates across a number of kernels as follows:

1. The prach_compute_correlation kernel takes input IQ data and performs averaging among repetitions followed by a time-domain correlation (done in frequency domain) against a reference version of the expected PRACH signal. This kernel simultaneously operates on each PRACH occasion.

2. An inverse FFT kernel transforms the frequency domain correlation results to time domain. A separate kernel operates on each occasion.

3. The prach_compute_pdp kernel performs non-coherent combining of correlation results for each preamble zone. It then calculates power and the peak index and value for each preamble zone.

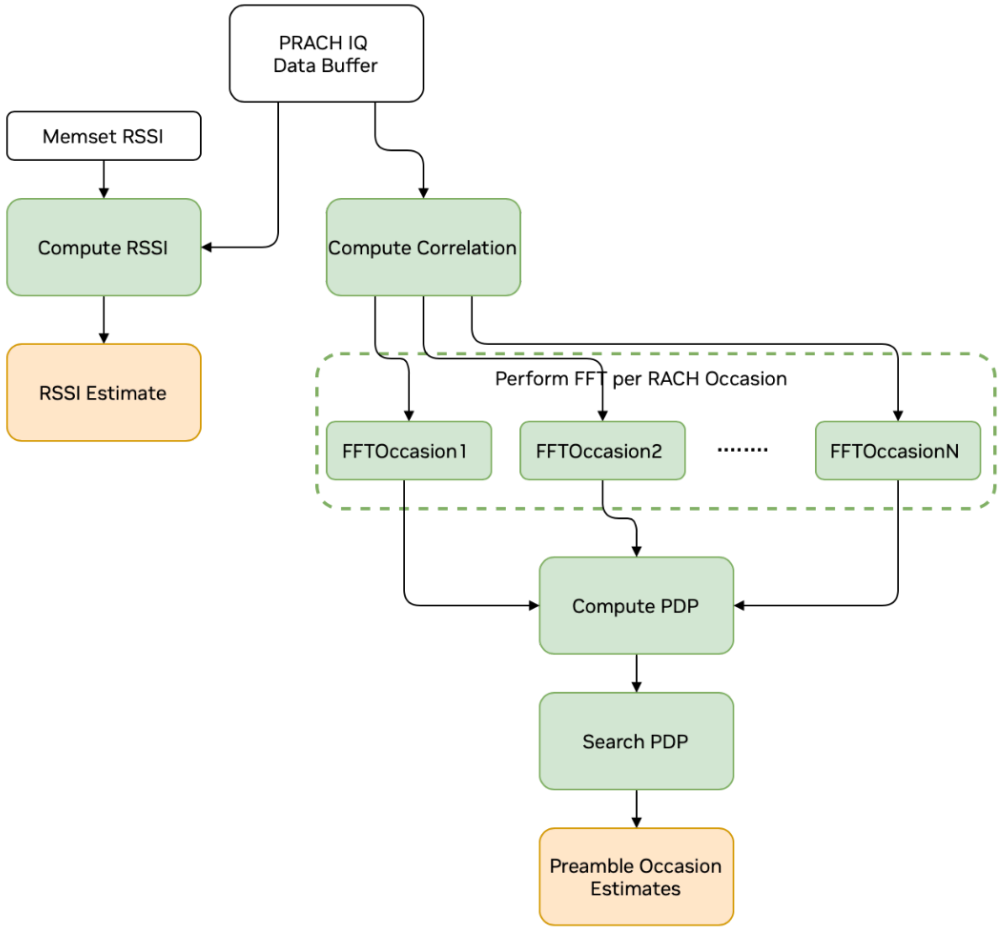
4. The prach_search_pdp kernel computes preamble and noise power estimates and reports the preamble index with peak power. It also does threshold-based detection declaration.

There is also a separate set of kernels as part of the PRACH pipeline for performing RSSI calculations.

1. The memsetRssi kernel clears a device buffer used in computing RSSI.

2. The prach_compute_rssi kernel computes RSSI for each PRACH occasion both for each antenna and average power over all antennas

3. The memcpyRssi kernel stores the RSSI results in host-accessible memory



Graph Diagram of the PRACH Pipeline

Input Buffer	PrachRx:: h_dynParam[i].dataRx
Data type	CUPHY_C_16_F : tensor for each occasion buffer
Dimensions	[(Preamble length+5)*Number of repetitions , N_ant]
Output Buffer	PrachRx:: numDetectedPrmb
Data type	CUPHY_R_32U : tensor vector of uint32
Dimensions	[1, PRACH_MAX_OCCASIONS_AGGR]
Description	Number of detected preambles for each occasion
Output Buffer	PrachRx:: prmbIndexEstimates
Data type	CUPHY_R_32U : tensor vector of uint32

Dimensions	[PRACH_MAX_NUM_PREAMBLES, PRACH_MAX_OCCASIONS_AGGR]
Description	Detected preamble index for each preamble and occasion
Output Buffer	PrachRx:: prmbDelayEstimates
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[PRACH_MAX_NUM_PREAMBLES, PRACH_MAX_OCCASIONS_AGGR]
Description	Delay estimate for each preamble and occasion
Output Buffer	PrachRx:: prmbPowerEstimates
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[PRACH_MAX_NUM_PREAMBLES, PRACH_MAX_OCCASIONS_AGGR]
Description	Power estimate for each preamble and occasion
Output Buffer	PrachRx:: antRssi
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[N_ant, PRACH_MAX_OCCASIONS_AGGR]
Description	RSSI for each antenna and occasion
Output Buffer	PrachRx:: rssi
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[1, PRACH_MAX_OCCASIONS_AGGR]
Description	RSSI for each occasion
Output Buffer	PrachRx:: interference
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[1, PRACH_MAX_OCCASIONS_AGGR]
Description	Interference for each occasion
Output Buffer	PrachRx:: prmbPowerEstimates
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[PRACH_MAX_NUM_PREAMBLES, PRACH_MAX_OCCASIONS_AGGR]

Description	Power estimate for each preamble and occasion
Output Buffer	PrachRx:: antRssi
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[N_ant, PRACH_MAX_OCCASIONS_AGGR]
Description	RSSI for each antenna and occasion
Output Buffer	PrachRx:: rssi
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[1, PRACH_MAX_OCCASIONS_AGGR]
Description	RSSI for each occasion
Output Buffer	PrachRx:: interference
Data type	CUPHY_R_32_F : tensor vector of float values
Dimensions	[1, PRACH_MAX_OCCASIONS_AGGR]
Description	Interference for each occasion

Performance Optimization

The cuPHY library is designed to accelerate PHY layer functionality of commercial grade 5G gNB DU. Software optimizations ensure reduced latency and scalable performance with the increased number of cells. We can categorize them as:

- Use of CUDA Graphs: The cuPHY library makes use of CUDA graph feature to reduce kernel launch latency. The CUDA kernels implementing signal processing components within each cuPHY physical layer channel pipeline are represented as nodes in a CUDA graph and the inter-component dependencies as edges between nodes. Since graph creation is expensive, a base graph with the worst case topology is created during initialization of channel pipelines where there are several specializations of component kernels. When the channel is scheduled for a given slot only the necessary subset of graph nodes are updated and enabled.

- Use of MPS (Multi-Process Service): The cuPHY driver creates multiple MPS contexts, each with an upper limit to the maximum number of SMs (Streaming Multiprocessors) that can be used by kernels launched there. MPS contexts for control channels (e.g. PUCCH, PDCCH) usually have significantly lower SM limits

compared to MPS contexts for shared channels due to the expected computation load. Each MPS context also has one or more CUDA streams associated with it, with potentially different CUDA stream priorities.

- Kernel fusion: the cuPHY implementation may fuse functionality from different processing steps into a single CUDA kernel for improved performance. For example, the rate matching, scrambling and modulation processing steps of the downlink shared channel are all performed in a single kernel. The motivation for these customizations is to reduce memory access latency and therefore improve performance. For example, assume that there are two kernels that are run in sequence. The first kernel makes a computation, writes the output to the global memory and the second kernel needs to read this output from the global memory to continue the computation. In this case, fusing these two kernels can reduce the number of accesses to the global memory, which has higher latency.

- Optimization of L1-L2 data flow: Data flow between the L2 and L1, and between the L1 and the FH are important for optimization of the latency. Data TB payloads for PDSCH channel need to be copied from L2 to L1 whenever a PDSCH channel is scheduled by the L2. The size of TBs increases with higher data throughput and the number of TBs also can also increase with the number of cells and the number of UEs scheduled on a given time slot. cuPHY library pipelines the TB H2D (host to device) copy to run in parallel with PDSCH channel setup processing. Such pipelining hides the TB H2D copy latency reducing overall PDSCH completion time.

Running cuPHY Examples

cuPHY library comes with example programs that can be used to test cuPHY channel pipelines and components. How to run cuPHY channel pipelines are explained in Aerial Release Guide Document in the section “Running the cuPHY Examples”. Please refer to the release guide on how to run the cuPHY channel pipelines. In running these examples, note that recent cuPHY implementation uses graphs mode to improve performance as explained in Section 3.1 of this document.

cuPHY library also includes examples for its components. Some examples are provided below.

Uplink channel estimation

```
cuPHY/build/examples/ch_est/cuphy_ex_ch_est -i ~/<tv_name>.h5
```

Sample test run:

```
cuPHY/build/examples/ch_est/cuphy_ex_ch_est -i TVnr_7550_PUSCH_gNB_CUPHY_s0p0.h5 UE group 0: ChEst SNR: 138.507 dB ChEst test vector TVnr_7550_PUSCH_gNB_CUPHY_s0p0.h5 PASSED 22:53:17.726075 datasets.cpp:974 WRN[90935 ] [CUPHY.PUSCH_RX] LDPC throughput mode disabled 22:53:17.943272 cuphy.hpp:84 WRN[90935 ] [CUPHY.MEMFOOT]cuphyMemoryFootprint - GPU allocation: 684.864 MiB for cuPHY PUSCH channel object (0x7ffc16f09f90). 22:53:17.943273 pusch_rx.cpp:1188 WRN[90935 ] [CUPHY.PUSCH_RX] PuschRx: Running with eqCoeffAlgo 3
```

Simplex decoder

```
cuPHY/build/examples/simplex_decoder/cuphy_ex_simplex_decoder -i ~/<tv_name>.h5
```

Sample test run:

```
cuPHY/build/examples/simplex_decoder/cuphy_ex_simplex_decoder -i TVnr_61123_SIMPLEX_gNB_CUPHY_s0p0.h5 AERIAL_LOG_PATH unset Using default log path Log file set to /tmp/simplex_decoder.log 22:57:29.115870 WRN 92956 0 [NVLOG.CPP] Using /opt/nvidia/cuBB/cuPHY/nvlog/config/nvlog_config.yaml for nvlog configuration 22:57:33.455795 WRN 92956 0 [CUPHY.PUSCH_RX] Simplex code: found 0 mismatches out of 1 codeblocks Exiting bg_fmtlog_collector - log queue ever was full: 0
```

PUSCH de-rate match

```
cuPHY/build/examples/pusch_rateMatch/cuphy_ex_rateMatch -i ~/<tv_name>.h5
```

Sample test run:

```
cuPHY/build/examples/pusch_rateMatch/cuphy_ex_pusch_rateMatch -i TVnr_7143_PUSCH_gNB_CUPHY_s0p0.h5 AERIAL_LOG_PATH unset Using default log
```

```
path Log file set to /tmp/pusch_rateMatch.log 22:58:20.673934 WRN 93384 0 [NVLOG.CPP] Using cuPHY/nvlog/config/nvlog_config.yaml for nvlog configuration 22:58:20.896254 WRN 93384 0 [CUPHY.PUSCH_RX] LDPC throughput mode disabled nUes 1, nUeGrps 1 nMaxCbsPerTb 3 num_CBs 3 uciOnPuschFlag OFF nMaxTbs 1 nMaxCbsPerTb 3 maxBytesRateMatch 156672 22:58:21.037299 WRN 93384 0 [CUPHY.MEMFOOT] cuphyMemoryFootprint - GPU allocation: 684.864 MiB for cuPHY PUSCH channel object (0x7ffe23b0f690). 22:58:21.037302 WRN 93384 0 [CUPHY.PUSCH_RX] PuschRx: Running with eqCoeffAlgo 3 22:58:21.037810 WRN 93384 0 [CUPHY.PUSCH_RX] detected 0 mismatches out of 65280 rateMatchedLLRs Exiting bg_fmtlog_collector - log queue ever was full: 0
```