White Paper

# Broadcom® Congestion-Aware Sprayed Traffic Feature

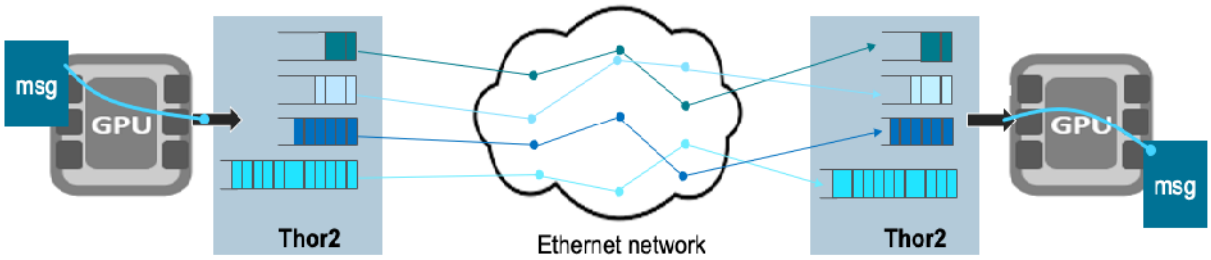
## Achieving Congestion-Aware Traffic Distribution Using Standard RoCEv2

## Introduction

Network throughput and latency are significant factors in the performance that can be achieved by the collective operations commonly used in AI/ML applications. Network performance is impacted by the efficiency at which all the available paths in the network can be utilized. If one network path is congested, it can create a bottleneck that impacts the overall application performance.

The Broadcom® Congestion-Aware Sprayed Traffic (CAST) feature dynamically distributes traffic across network paths to more efficiently use available bandwidth and minimize network congestion. This feature can lower latency and improve application performance for large AI clusters using leaf and spine fabrics for scale-out traffic.

**Figure 1: Illustration of CAST Concept**



# Background

RDMA over Converged Ethernet (RoCE) networks are often used in AI/ML deployments. In RoCE networks, Queue Pairs are used to provide point-to-point connectivity between two endpoints. The RoCE traffic typically traverses a multi-tier Ethernet fabric, containing top-of-rack (ToR) and spine switches, that provide the connectivity between endpoints. One mechanism that is commonly used to distribute the traffic across multiple paths is equal-cost multipath (ECMP) routing. ECMP is typically performed independently by each switch.

ECMP uses an algorithm to distribute the traffic. Multiple algorithms have been implemented. A popular algorithm, that can maintain packet order, is to perform a hash on packet header fields, such as the {source IP address, destination IP address, protocol, source port number, destination port number} 5-tuple, and then use the hash result to select one of the available paths to forward the packet.

ECMP is effective, but leaves room for improvement. When traffic patterns are composed of a relatively small number of very high bandwidth flows, even a small number of collisions (mapping multiple flows to a common path) can cause severe congestion. One improvement that has been implemented by Collective Communication Libraries (CCLs) is to establish multiple Queue Pairs between two endpoints. The CCL then distributes the point-to-point traffic load across the multiple Queue Pairs, effectively spreading a large message transfer across multiple network paths.

For example, two mechanisms have been implemented in the NVIDIA CCL (NCCL) and the AMD ROCm CCL (RCCL):

- Round-robin (RR) load balancing – Send the first request over one Queue Pair, send the next request over another Queue Pair, and so on.
- Striping load balancing – Break the request into equal-sized subchunks and then transmit one subchunk over each Queue Pair.

Such Queue Pair load balancing provides improvement over ECMP alone, but also leaves room for additional improvement.

# Congestion-Aware Sprayed Traffic Overview

The Broadcom CAST feature improves Queue Pair load balance through the addition of weighted scheduling. The idea is to use round-trip time (RTT) measurements to schedule more traffic over the Queue Pairs that are experiencing the least congestion. The RTT measurements are used to calculate weights for a weighted scheduler. CAST can be applied to either of the CCL load balancing (LB) mechanisms listed above: round robin or striping.

- Queue Pair Striping LB Scheduler = CCL striping load balancing enabled with CAST
- Queue Pair RR LB Scheduler = CCL round-robin load balancing enabled with CAST

First, how this is done in conjunction with Queue Pair Striping is described. Then the concept is extended to the Queue Pair RR LB mechanism.

A number of metrics are possible based on the measured RTTs; including, but not limited to, the following measurements:

- Raw RTT
- (RTT – transmission time of message payload)
- (RTT – transmission time of message payload – minimum latency for the network)
- (RTT – transmission time of message payload – minimum latency for the network - transmission time of protocol headers associated with the message)

For CAST, the following metric was selected: (RTT – transmission time of message payload – minimum latency for the network)

Once the metric is selected and computed, an estimator function is applied. The estimator function filters the instantaneous metric values to form a more stable estimate of network congestion that is then used by the Queue Pair scheduler. A number of estimator functions are possible; including, but not limited to, the following functions:

- Average:
  - Estimate = (sum of metric values) / (number of measurements)
  - The average history (that is, sum and measurement count) may be reset to zero at a configured interval to increase responsiveness to changing network conditions
- Weighted average:
  - Estimate = (metric value × weight) + (estimate × (1 – weight))
  - Weighted average can respond more quickly to metric changes
  - TCP previously used a weighted average estimator with a weight of 0.1
- TCP estimator:
  - Estimate = estimate + (0.125 × (metric value – estimate))
  - The TCP estimator can respond aggressively to metric changes

The current CAST version supports the following estimator functions:

- Average
- Weighted average

Conceptually, the estimates are used to compute scheduler weights as shown in the following text:

```
for (qp = 0, sum = 0; qp < number of qp's; qp++) {
   new_estimate[qp] = 1.0 / estimate[qp];
   sum += new_estimate[qp];
}
sched_weight[qp] = new_estimate[qp] / sum;
```

However, based on implementation experience, it was determined that the RTT measurements contain considerable jitter that were adversely impacting CAST effectiveness. The RTT measurement is made by CCL software as follows:

- The RTT measurement begins when a transmission request is submitted to a Queue Pair using the IB Verb API.
- The RTT measurement ends when the associated transmit completion is processed by the CCL software as a result of a polling operation.

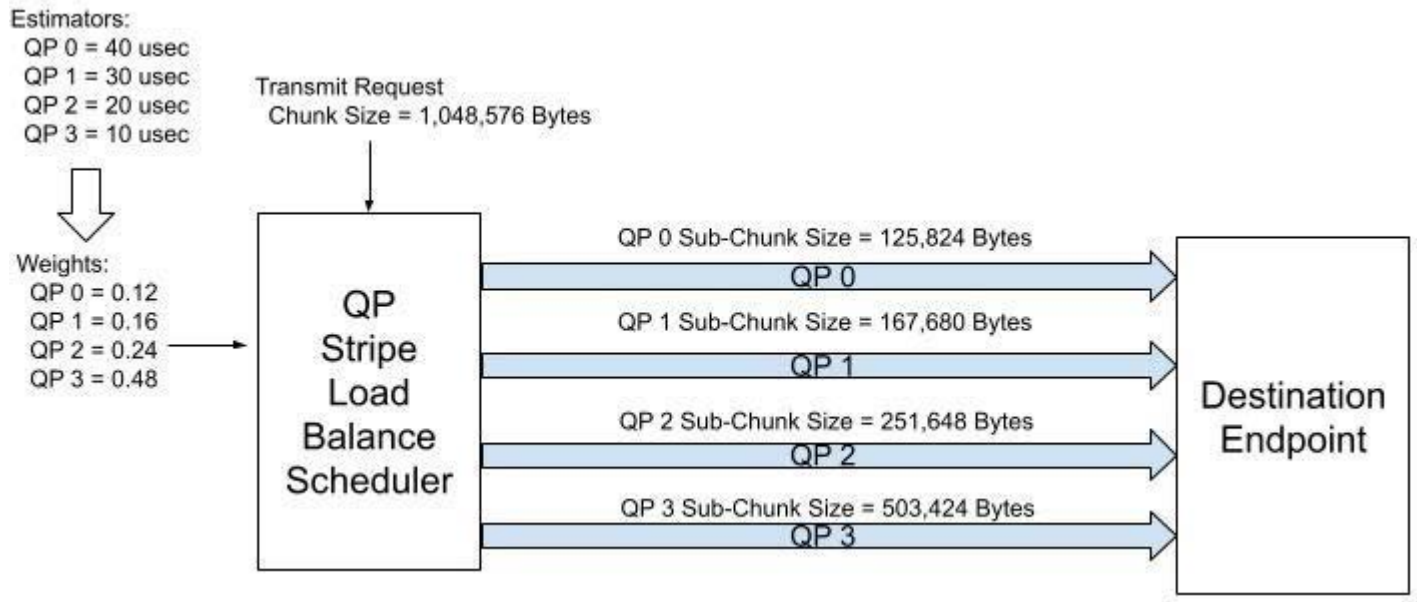
Consequently, in an effort to filter the jitter bias, the following algorithm is used to compute the scheduler weights:

```
for (qp = 0, min_estimate = DBL_MAX; qp < number of qp's; qp++) {
   if (estimate[qp] < min_estimate) {
       min_estimate = estimate[qp];
       min_metric = min_metric[qp];
   }
}
for (qp = 0, sum = 0; qp < number of qp's; qp++) {
   new_estimate[qp] = 1.0 / ((estimate[qp] - min_estimate) + min_metric);
   sum += new_estimate[qp];
}
    sched_weight[qp] = new_estimate[qp] / sum;
```

The data plane uses the scheduler weights to determine the subchunk size to be sent on the associated Queue Pair. This changes the distribution of the subchunk size transmitted on the Queue Pairs from an even distribution to a weighted distribution. The scheduler weights used by the data plane may be refreshed periodically according to a configured update interval.

Operation of the Queue Pair Stripe LB scheduler is shown in the following figure. In this example, subchunk sizes for QP0, QP1, and QP2 are rounded down to the next lowest multiple of 128 bytes.

**Figure 2: Queue Pair Stripe LB Scheduler**

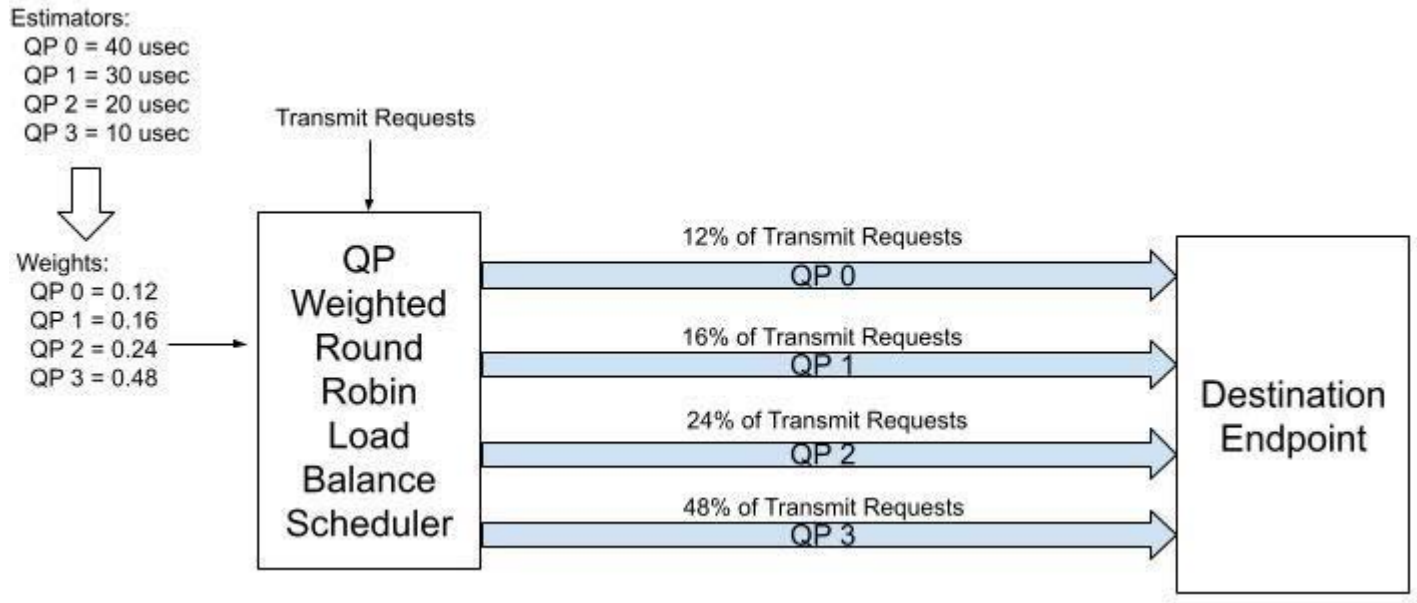


The Queue Pair Stripe LB scheduler operation can be enhanced by enforcing a threshold that constrains the minimum subchunk size. Since there are overheads, such as completion processing time, associated with transmitting a subchunk, there is a point at which further division of the transmit request chunk size into smaller subchunks is not advantageous from a performance perspective. CAST limits the number of Queue Pairs used by the scheduler according to the following algorithm:

```
sub-chunk size = chunk size / number of QPs available;
if (sub-chunk size < sub-chunk threshold)
   number of QPs used = 1;
else
   number of QPs used = number of QPs available;
```

As mentioned previously, the estimates can also be used to implement a Queue Pair weighted round-robin (WRR) LB scheduler. In this case, the weight associated with a Queue Pair indicates the percentage of the transmit requests that should be sent on the Queue Pair as shown in the following figure.

**Figure 3: Queue Pair WRR LB Scheduler**




When the Queue Pair Stripe LB scheduler falls back to the use of one Queue Pair due to the subchunk threshold, a configuration parameter controls the use of RR or WRR scheduling.

# Configuration Parameters

Operation of the CAST feature is controlled by the environment variables described in the following table.

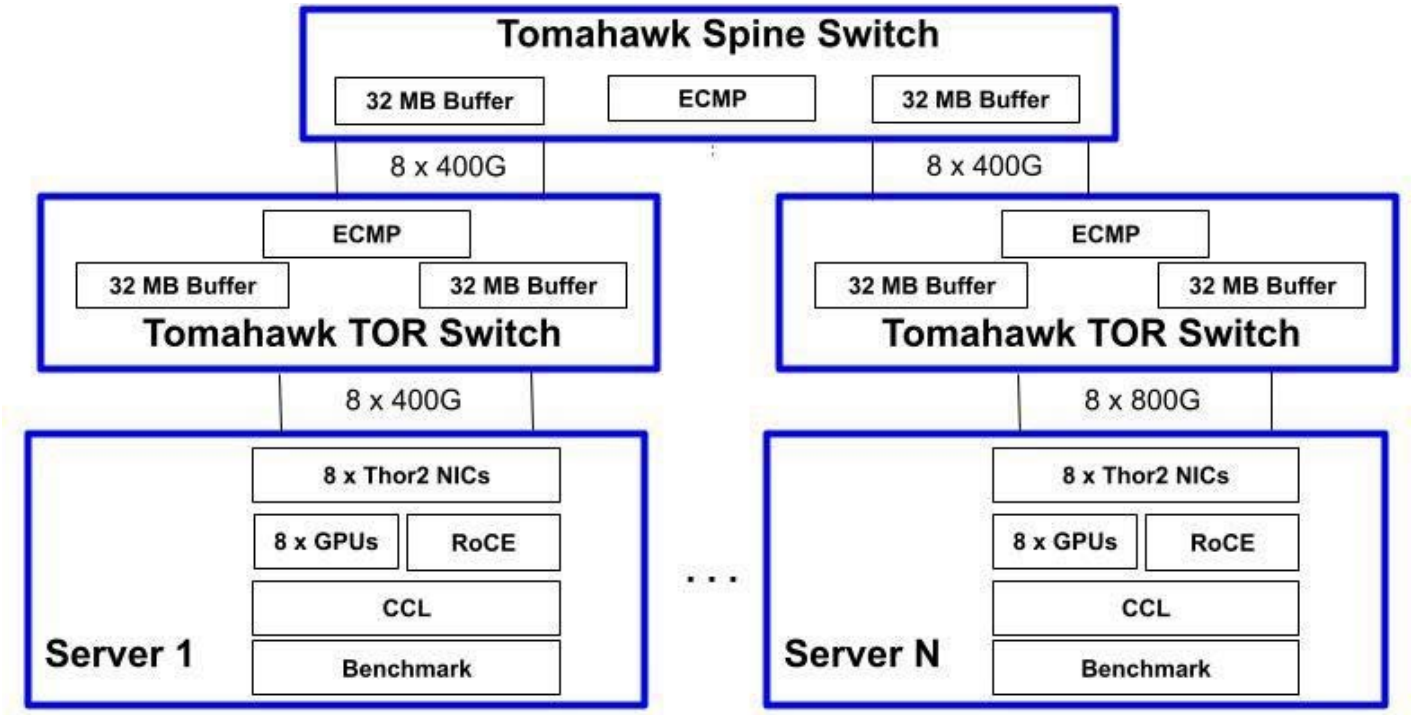
**Table 1: CAST Configuration Environment Variables**

| Environment Variable | Description |
|---|---|
| NCCL_IB_QP_SCHED_ENABLE | 1 => enable the feature, defaults to 0 |
| NCCL_IB_QP_SCHED_RESET_INTERVAL | The interval where the RTT history is reset in units of milliseconds.<br>■ default = 60000<br>■ 0 => never reset |
| NCCL_IB_QP_SCHED_UPDATE_INTERVAL | The interval where scheduling weights are refreshed in units of microseconds, default = 50. |
| NCCL_IB_QP_SCHED_WEIGHT | 0 => use average estimator<br>Otherwise, use the floating point weight that is applied to the RTT sample in the weighted average estimator, default = 0. |
| NCCL_IB_QP_SCHED_SPLIT_DATA_MIN | The minimum amount of data per Queue Pair required to split the data transmission across multiple Queue Pairs. If the minimum is not met, the data is transmitted on a single Queue Pair, default = 64K bytes. |
| NCCL_IB_QP_SCHED_WRR_ENABLE | 1 => enable the feature, defaults to 0<br>If enabled, weighted RR scheduling is performed instead of the standard RR scheduling when the NCCL_IB_QP_SCHED_SPLIT_DATA_MIN condition is not satisfied. |

# CAST Test Topology

The CAST concept was tested using the topology shown in the following figure.

**Figure 4: Broadcom CAST Test Topology**



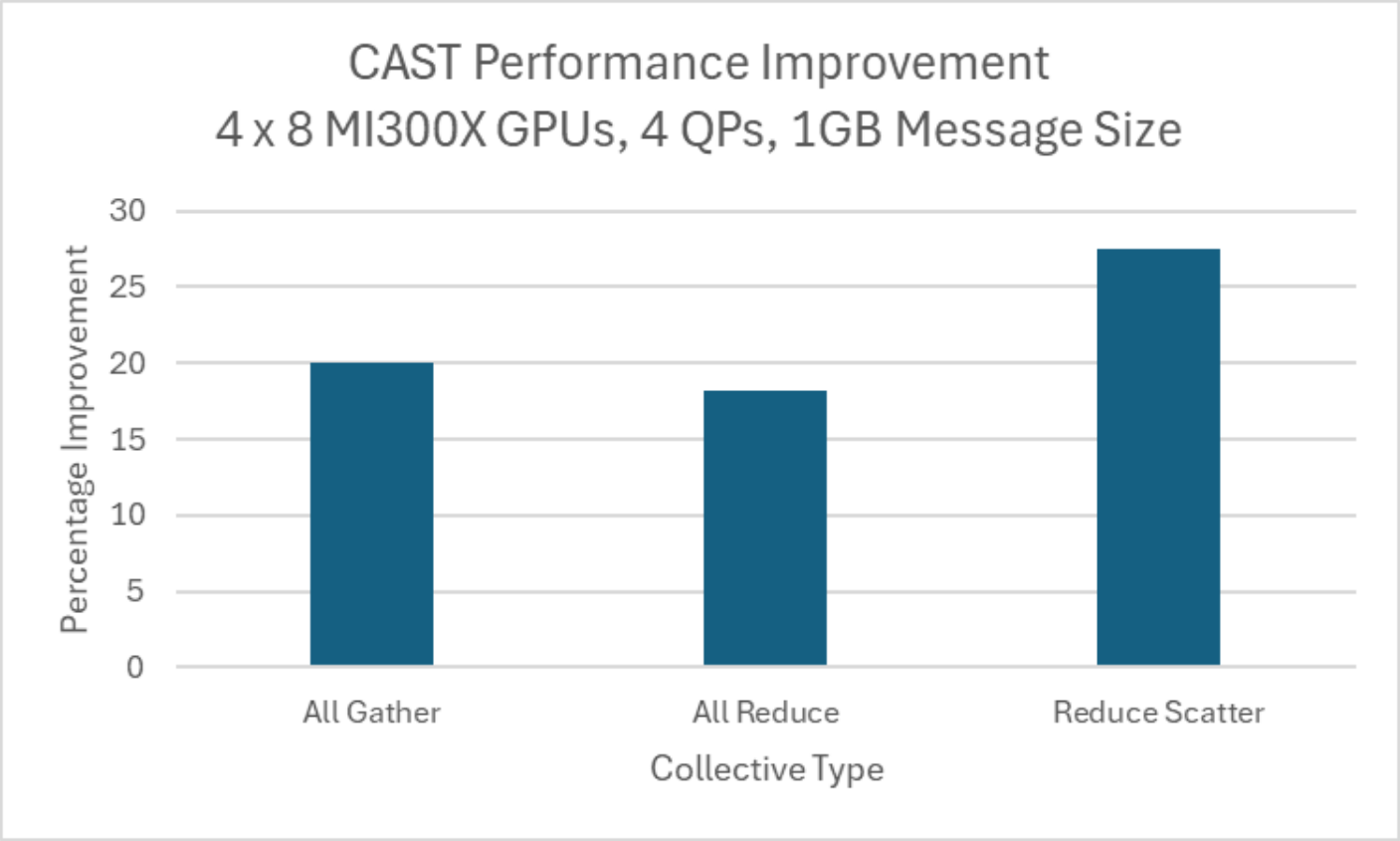

The test topology is designed to emulate the ECMP congestion that would be seen in larger clusters. The ToR switches forward all upstream traffic to the spine, which promotes ECMP-based congestion at the spine. A single Tomahawk® switch can be partitioned into multiple virtual ToR switches to minimize the equipment needed to implement the test topology.

# Performance Results

The following figure shows the CAST performance improvement for three collectives when using a 1-GB message size with four Queue Pairs per connection. The improvement is expressed as the percentage that collective throughput increased when CAST was enabled. The performance data was collected on a cluster with four server nodes, where each node was connected to a virtual ToR switch, and each node was equipped with the following components:

- 8 × AMD MI300X GPUs
- 8 × Broadcom Thor2 NICs operating at 400Gb/s

**Figure 5: Example of CAST Performance Improvement**

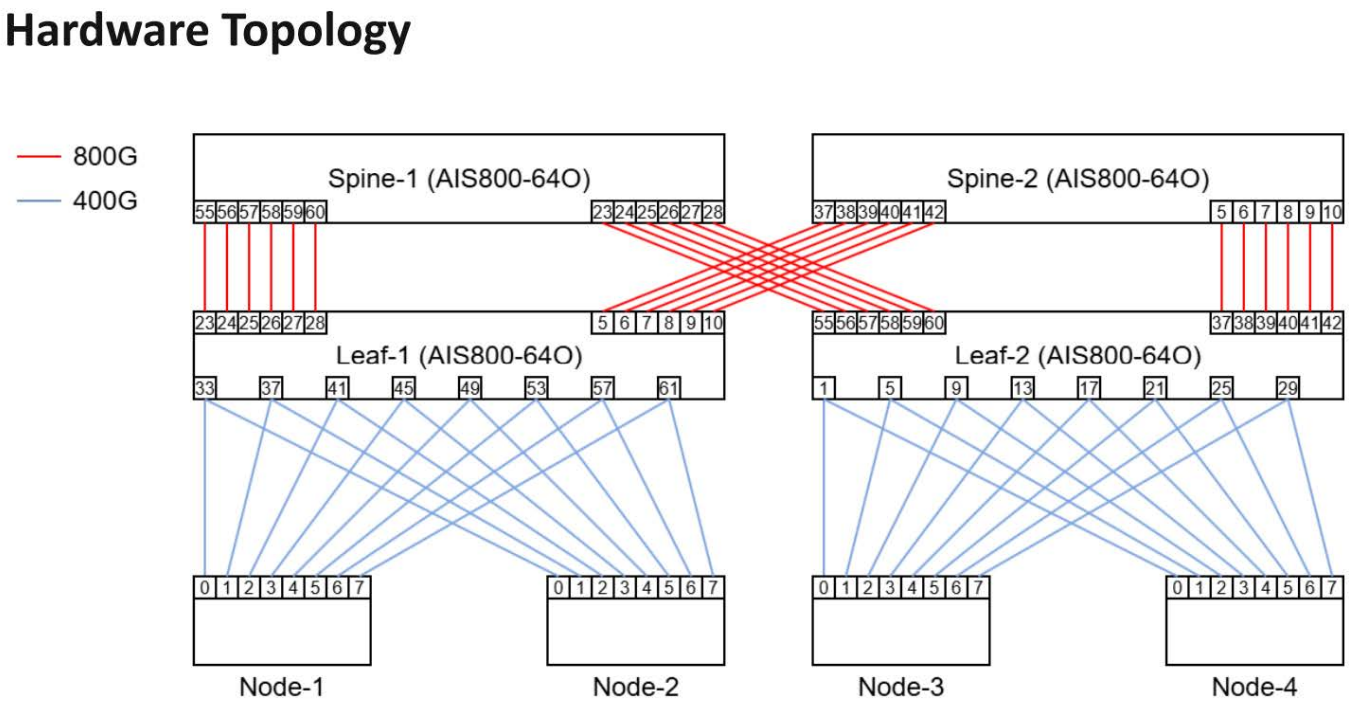


# Edgecore Testing

The CAST feature was tested using the topology shown in the following figure. The leaf switches use virtual routers and Border Gateway Protocol to set up a logical topology that sends all traffic to the spine switches to emulate a larger cluster.

The leaf and spine switches are Edgecore AIS800-64O Tomahawk 5 Ethernet switches. Each node includes eight Thor2 400G Ethernet NICs attached, one-to-one.

**Figure 6: Edgecore CAST Test Topology**




## Edgecore Performance Results

The following figures compare the results using RCCL's collective benchmark tests for various collective operations. All use the network topology shown in the previous figure. Results are shown for four iterations per collective operation:

- split-no-cast = Queue Pair split LB without CAST
- split-cast = Queue Pair split LB scheduler using CAST
- wrr-no-cast = Queue Pair round-robin LB without CAST
- wrr-cast = Queue Pair round-robin LB scheduler using CAST

The results for the out-of-place option in the benchmark are shown. This indicates the operation reads from an input buffer and writes the result to a different output buffer. The message size refers to the overall collective. The size of the data movement on each GPU is scaled based on the total number of GPUs and the type of operation.

**Figure 7: AllReduce Performance Using CAST**

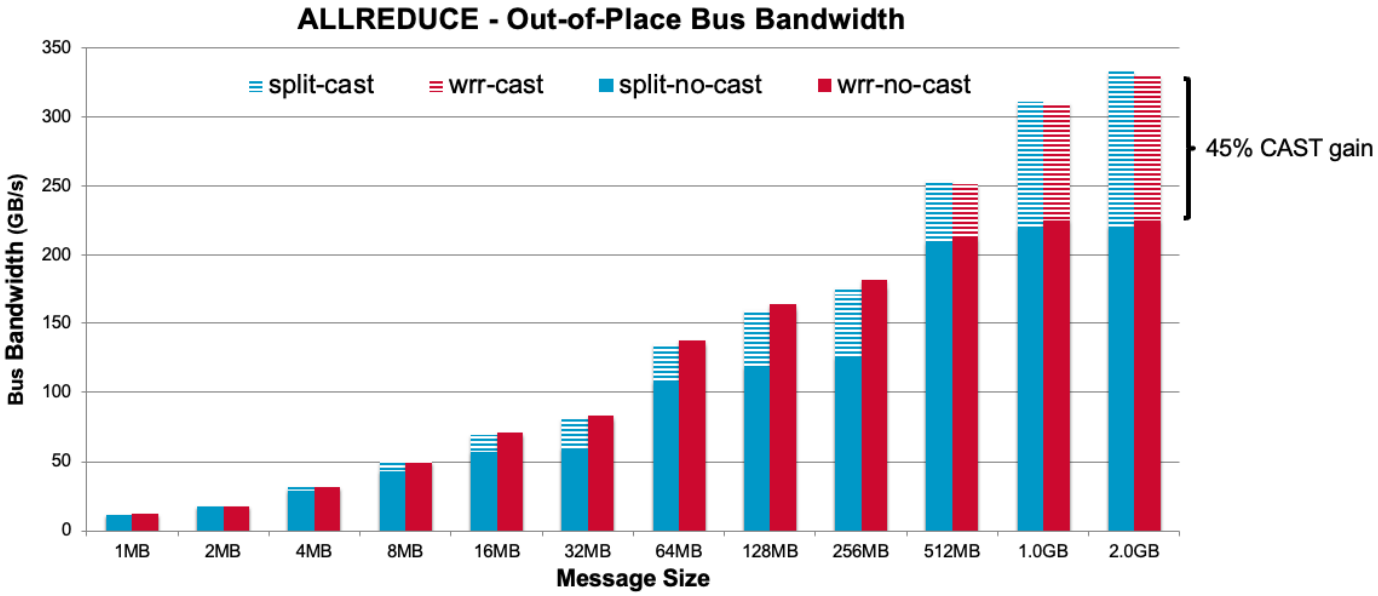



**Figure 8: AllGather Performance Using CAST**

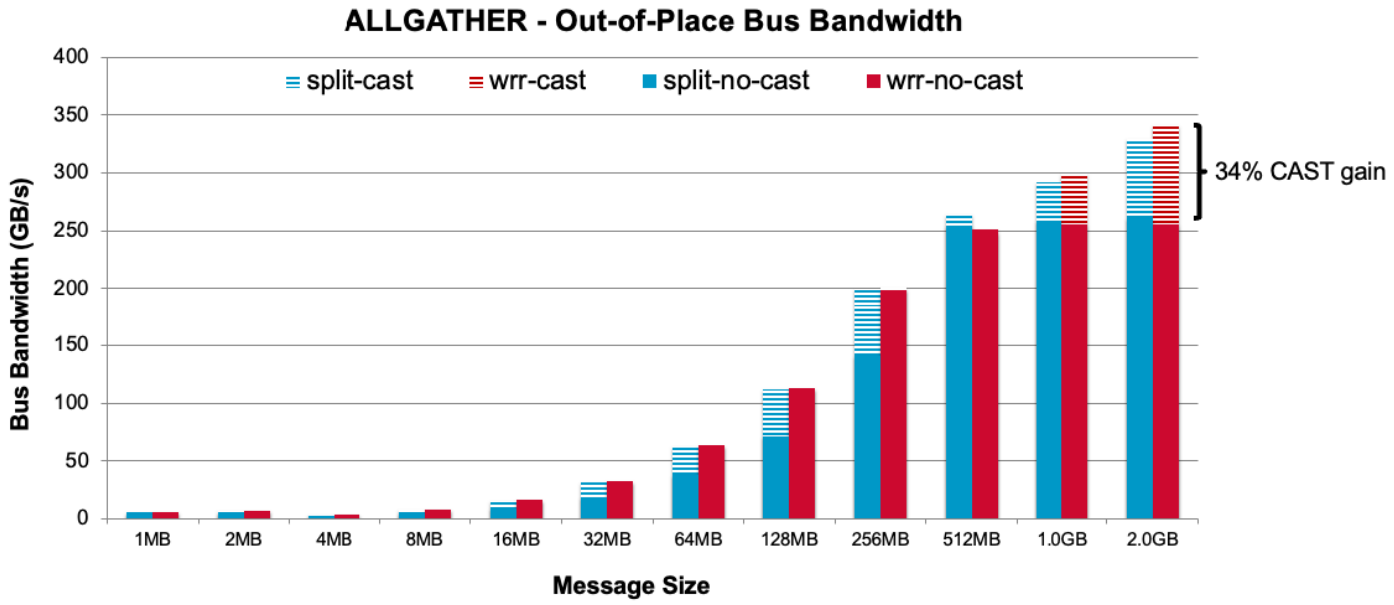


**Figure 9: AlltoAll Performance Using CAST**

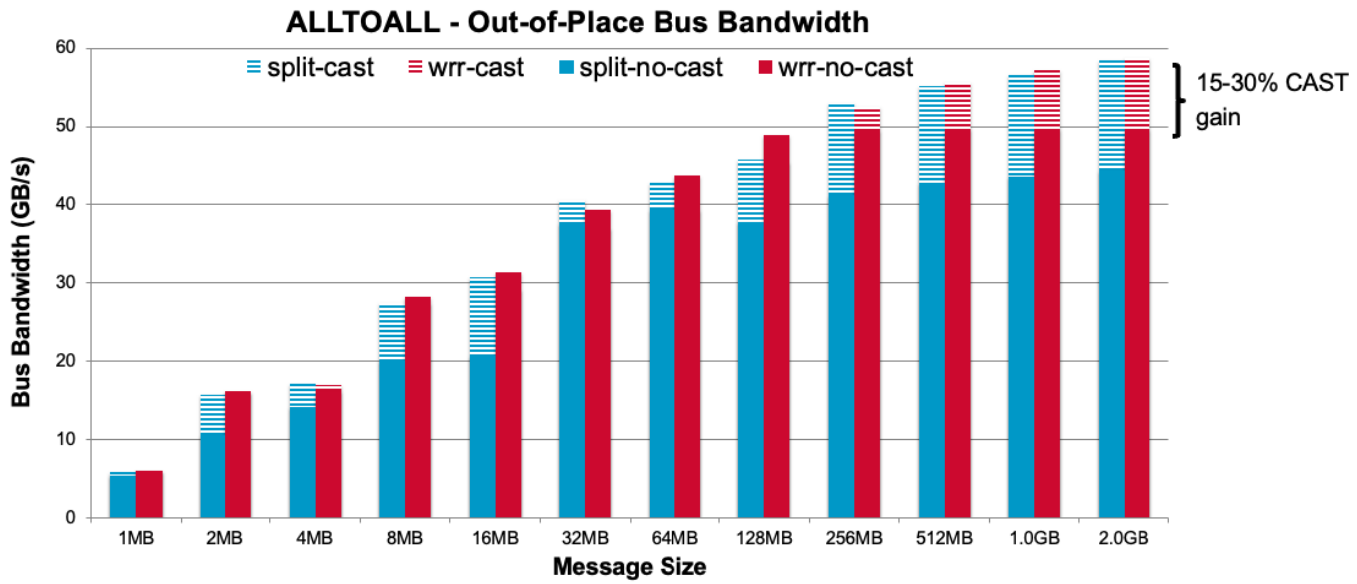
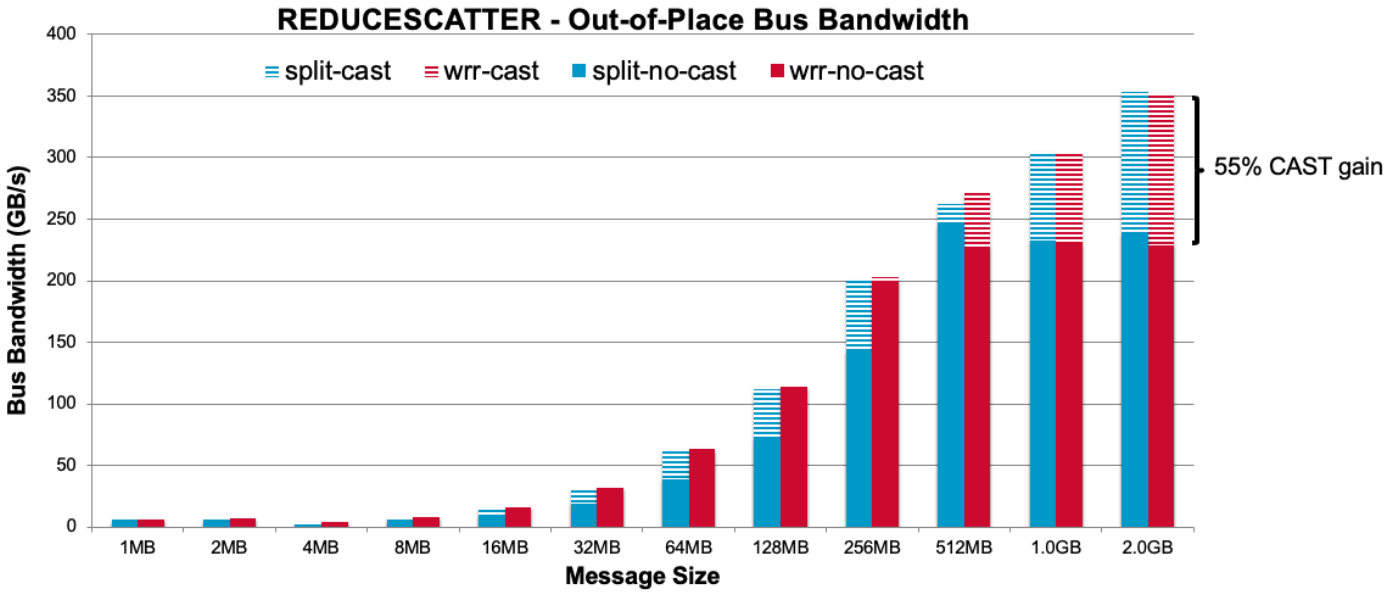



**Figure 10: ReduceScatter Performance Using CAST**

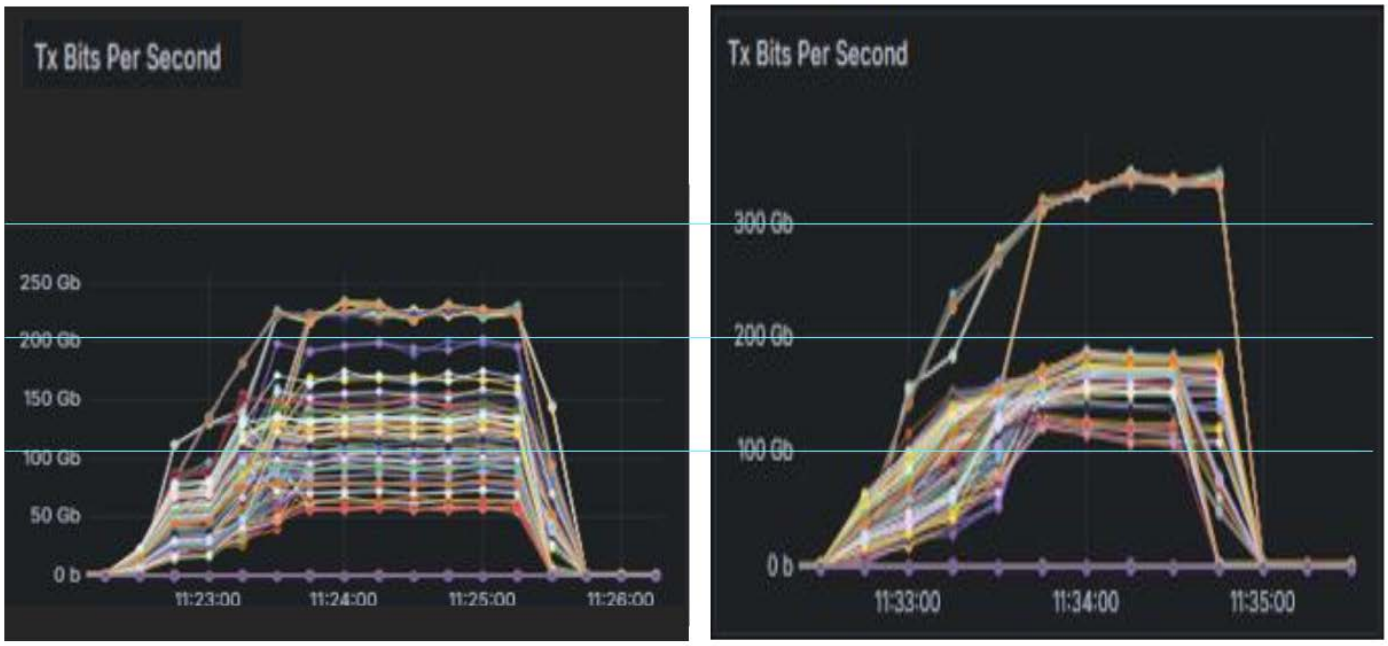


# Summary of Results

CAST can improve performance on CCL collective operations on AI clusters using a leaf-spine topology. RCCL bus bandwidth may be increased from 10% to up to 50% for large messages.

Fat tree topologies with two or more tiers of switching may experience ECMP collisions, particularly with traffic patterns using a relatively small number of high bandwidth flows. CAST distributes data across multiple network paths to more evenly distribute traffic and thereby optimize network performance. The following figure shows the bandwidth per network path without CAST and with CAST enabled.

**Figure 11: AllReduce Average Bus Bandwidth with and without CAST**




AllReduce without CAST
(average bus BW: 101.48Gb/s)

AllReduce with CAST
(average bus BW: 135.33Gb/s)

Workloads that use larger messages tend to experience more network congestion and therefore benefit more from using CAST. CAST can be less effective with smaller collective message sizes for multiple reasons:

- Smaller message sizes produce less network congestion.
- Smaller message sizes produce smaller transmit request chunk sizes that must not exceed the CAST subchunk threshold so that CAST scheduling is not applied.

Performance improvement is seen across all collective operations.

# Implementation Overview

The CAST implementation is based on the CCL Network Plugin capability that enables customization of the network transport layer. The implementation starts by copying the network transport layer implementation in the `src/transport/net_ib.cc` file and using it as a base. A high-level overview of the modifications that were made to the base code is provided in this section with the goal of making it easier to read and understand the code.

The CCL network plugin code is in the `ext-net/net_ib_bnxt` directory. The contents of the `ext-net/net_ib_bnxt` directory for NCCL is shown below:

```
root@hpci5201:/mnt/irvnas/home/sandbox/developer/x86/nccl/nccl_v2.26.2-1/ext-net/net_ib_bnxt# ls
libnccl-net-bnxt.so Makefile net_ib_bnxt.cc
```

The network plugin source code is in the `net_ib_bnxt.cc` file. Type `make` to build the code and produce the `libnccl-net-bnxt.so` library. On NCCL systems, the `libnccl-net-bnxt.so` library is then copied to the `../../build/lib/libnccl-net-bnxt.so` location.

The contents of the `ext-net/net_ib_bnxt` directory for RCCL is similar, but the library name is changed to `librccl-net-bnxt.so`. On RCCL systems, the `librccl-net-bnxt.so` library is then copied to the `../../build/librccl-net-bnxt.so` location. There is also one additional file in the `ext-net/net_ib_bnxt` directory for RCCL, `net_plugin_tuner_api.h`. This file defines an API that enables a tuner plugin to dynamically modify CAST parameters on a `{collective type, message size}` basis.

The environment variables used to invoke the network plugin are slightly different for NCCL versus RCCL as shown in the following text:

- NCCL:
  - NCCL_NET=”IB_BNXT”
  - NCCL_NET_PLUGIN=bnxt
- RCCL:
  - NCCL_NET=”IB_BNXT”
  - NCCL_NET_PLUGIN=librccl-net-bnxt.so

The code overview is presented as follows:

- New data structures
- Modified data structures
- New functions
- Modified functions

# New Data Structures

The new data structures are summarized in the following table.

**Table 2: Summary of New Data Structures**

| Data Structure Name | Description |
|---|---|
| ncclIbQpSchedParms | Parameters derived from the environment variable configuration |
| ncclIbQpTxData | Data about Queue Pair transmission |
| ncclIbRemapWrId | For remapping work, request an ID so that additional information is available at the completion time of sends |
| ncclIbQpTxStats | Statistics for scheduling Queue Pair transmissions |
| ncclIbQpTxSchedScratchpad | Scratchpad for computing scheduler weights |
| ncclIbQpTxSched | Scheduler for Queue Pair transmissions |
| ncclIbRrTokens | Tokens for the weighted RR Queue Pair scheduler |
| ncclIbRrQpTxSched | Scheduler for weighted RR Queue Pair transmissions |
| ncclIbQpSchedDesc | Queue Pair scheduling descriptor |

# Modified Data Structures

The modifications to existing data structures are summarized in the following table.

**Table 3: Summary of Data Structures Modifications**

| Data Structure Name | Description | Note |
|---|---|---|
| ncclIbRequest | Added:<br>■ `int ctsEvents[NCCL_IB_MAX_DEVS_PER_NIC];`<br>■ `struct ncclIbQpSchedDesc desc;` | `ctsEvents` is needed for enhanced completion processing, `desc` needed for fallback to RR or WRR scheduling |
| ncclIbSendFifo | Added:<br>■ `uint16_t rxReqIndex;` | Needed to remove binding between posted `recvs` and the `rx req` |
| ncclIbNetCommBase | Added:<br>■ `struct ncclIbRemapWrId remapWrId[MAX_REQUESTS];`<br>■ `struct ncclIbQpTxStats qpTxStats[NCCL_IB_MAX_QPS];`<br>■ `uint64_t nextQpTxStatsResetNs;`<br>■ `struct ncclIbQpTxSched qpTxSched[NCCL_IB_MAX_QPS];`<br>■ `struct ncclIbRrQpTxSched rrQpTxSched;`<br>■ `bool qpTxSchedInit;`<br>■ `uint64_t nextQpTxSchedUpdateNs;`<br>■ `int remapHead;`<br>■ `int rxPosts[NCCL_IB_MAX_QPS * NCCL_NET_IB_MAX_RECVS];` | Main data structures for the feature |

# New Functions

The new functions are summarized in the following table.

**Table 4: Summary of New Functions**

| Function Name | Description |
|---|---|
| initQpSchedParms | Initialize Queue Pair scheduling parameters based on environment variables |
| ncclIbGetRemap | Allocate remap data structure for enhanced completion processing |
| ncclIbFreeRemap | Free remap data structure |
| updateQpTxSched | Update scheduler weights |
| updateQpTxStats | Update RTT estimator |
| getEffectiveTxNqps | Determine number of Queue Pairs to used for transmit |
| ncclTunerPluginNotify | Implement tuner plugin API, RCCL only |

# Modified Functions

The modifications to existing functions are summarized in the following table.

**Table 5: Summary of Function Modifications**

| Function Name | Description |
|---|---|
| ncclIbAddEvent | Add support for tracking CTS completion events |
| ncclIbMultiSend | Add Queue Pair scheduling functionality |
| ncclIbIsend | Prepare for Queue Pair scheduling |
| ncclIbPostFifo | Add support for tracking CTS completion events |
| ncclIbIrecv | Remove binding between `wrap_ibv_post_recv` and `rx req` to support more flexible completion processing that enables fallback to RR or WRR scheduling |
| ncclIbTest | New completion handling that supports fallback to RR or WRR scheduling, and write immediate contents modified to identify when all `rx` completions have been received |

# Example Run-Line Command

The run-line command used to collect the AllGather performance data reported in Figure 5 is shown in the following example:

```
/root/irvnas_x86/benchmarks_build/ubuntu_22.04/openmpi_4.1.6_ucx_1.15.0/install/bin/mpirun --mca
orte_base_help_aggregate 0 -np 32 -host 1.1.10.15:8,1.1.11.15:8,1.1.12.15:8,1.1.13.15:8
--allow-run-as-root  --gmca btl_tcp_if_include eno8303 --gmca oob_tcp_if_include eno8303 --gmca btl
tcp,self  -x NCCL_IB_DISABLE=0 -x
NCCL_IB_HCA=bnxt_re0:1,bnxt_re1:1,bnxt_re2:1,bnxt_re3:1,bnxt_re4:1,bnxt_re5:1,bnxt_re6:1,bnxt_re7:  1
-x NCCL_IB_TC=104  -x NCCL_IB_GID_INDEX=3  -x NCCL_DEBUG=VERSION -x NCCL_IGNORE_CPU_AFFINITY=1 -x
NCCL_IB_QP_SCHED_ENABLE=1 -x NCCL_IB_SPLIT_DATA_ON_QPS=1 -x NCCL_IB_QPS_PER_CONNECTION=4  -x
NCCL_NET="IB_BNXT" -x NCCL_NET_PLUGIN=librccl-net-bnxt.so
/root/irvnas_x86/rccl/rccl-tests/build/all_gather_perf -b 1M -e 16G -f 2 -n 20 -w 5 -g 1 -c 1 -p 1 -t 1
```

BROADCOM®