Cascade Performance Evaluation and Tuning

Weijia Song

1 Abstract

This document contains Cascade/Derecho performance evaluation and tuning [2, 3].

2 How to choose window size

The *window_size* parameter in Derecho has not been fully discussed before. It controls the maximum number of messages sent in parallel. I designed a set of experiments to explore how *window_size* affect the performance of Cascade/Derecho.

2.1 RDMA queue depth and window size

The other two related parameters are tx_depth and rx_depth, which control the number of hardware buffer for RDMA. To understand how the those parameters affect Derecho/Cascade performance, I tested the throughput of *bandwidth_test* using three nodes in fractus with varying *window_size* and tx/rx_depth combination. To cover more cases, I also tested with 4 Kilo Byte and 1 Mega Byte message sizes to cover both SMC (4K) and RDMC (1M, with 256K RDMC *block_size*).

The first observation is that for both SMC and RDMC the hardware queue depth does not change the performance of *bandwidth_test* as long as it is bigger than a very small number. As shown in figure 1a and figure 1b, the performance is stable once the queue depth, shown as $TX/RXDepth$, is greater than 20. While in figure 1c and figure 1d, the perfromance is stable once the queue depth is greater than 10.

I found that the *window_size* parameter plays a critical role in performance optimization for *bandwidth_test*. For small messages sent with SMC, the throughput of *bandwidth_test* peaks when *window_size* is around 128 for one-sender setup (figure 1a) and 32 for all-sender setup (figure 1b). The corresponding peak throughputs are 330,000 ops for one-sender setup and 460,000 ops for all-sender setup. Obviously, 1MB messages requires less window to reach the peak performance. In one-sender setup, peak performance of 5,600 ops can be reached when *window_size* is between 2 and 32 (figure 1c). All-sender setup is very interesting because the peak performance of 11,000 ops declines along with growing *window_size*.

The same question arise in all other cases: while throughput declines when *window_size* grows beyond some point, no matter for SMC and RDMC?

2.2 Window size and message size

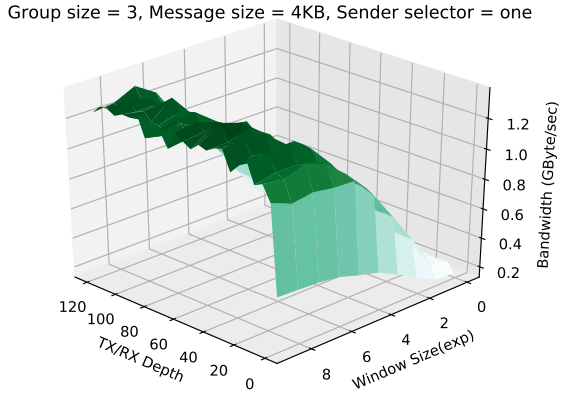
Msg Size in Bytes	One-sender Peak Thp in GB/s (*window_size*)	All-sender Peak Thp in GB/s *window_size*)
2K	0.442971 (2^7)	0.757996 (2^5)
4K	1.360827 (2^7)	1.862703 (2^6)
8K	2.796937 (2^8)	3.643267 (2^5)
16K	4.767933 (2^7)	6.195400 (2^5)
32K	5.754417 (2^8)	12.523633 (2^5)
64K	5.810907 (2^7)	16.852167 (2^7)
128K	5.647237 (2^6)	17.620333 (2^5)
256K	5.828473 (2^5)	17.804100 (2^5)
512K	5.924150 (2^7)	17.881900 (2^5)
1M	5.975967 (2^4)	17.940800 (2^5)

Table 1: SMC Window Size Choices

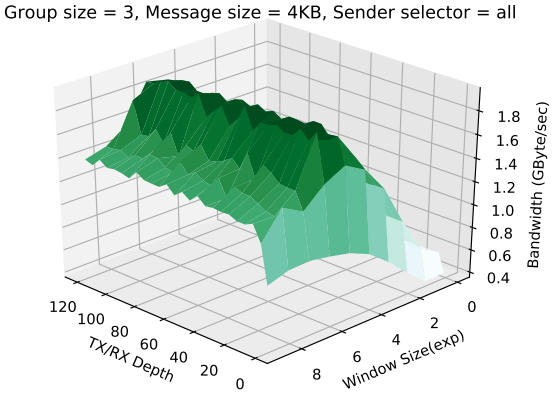
Msg Size in Bytes	One-sender Peak Thp in GB/s (*window_size*)	All-sender Peak Thp in GB/s *window_size*)
2K	0.088119 (2^2)	0.114299 (2^3)
4K	0.167284 (2^4)	0.221761 (2^3)
8K	0.328304 (2^8)	0.440711 (2^3)
16K	0.610523 (2^3)	0.854002 (2^3)
32K	1.051700 (2^6)	1.605157 (2^3)
64K	1.818327 (2^8)	2.939930 (2^3)
128K	2.780133 (2^3)	4.926673 (2^3)
256K	3.007057 (2^4)	5.846260 (2^2)
512K	4.450853 (2^1)	8.195923 (2^2)
1M	5.671763 (2^4)	10.174900 (2^1)
2M	6.644347 (2^2)	11.388600 (2^1)
4M	7.285793 (2^1)	12.173567 (2^1)
8M	7.627250 (2^2)	12.807933 (2^1)
16M	7.806543 (2^1)	13.326300 (2^1)
32M	7.911080 (2^1)	13.606667 (2^1)
64M	7.967493 (2^1)	13.841700 (2^1)

Table 2: RDMC Window Size Choices (block size is fixed at 256 KB)

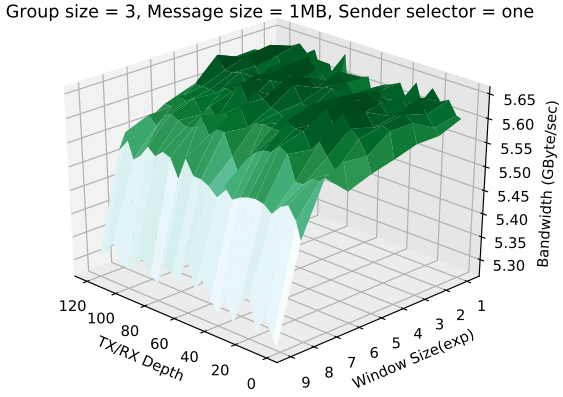
I did more experiments testing SMC and RDMC performance with varying *window_size* and message sizes. The results are shown in figure 2. We can see that the optimal *window_size* varies with message size as well as group size. Table 1 and table 2 show the peak throught of *bandwidth_test* I get from a limited *window_size* range.



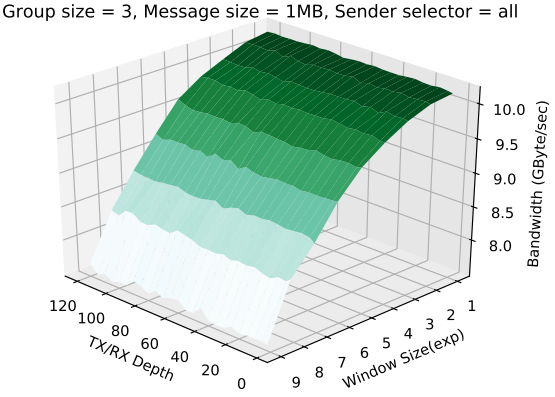
(a) SMC, One Sender, 4K Msg



(b) SMC, All Sender, 4K Msg



(c) RDMC, One Sender, 1M Msg



(d) RDMC, All Sender, 1M Msg

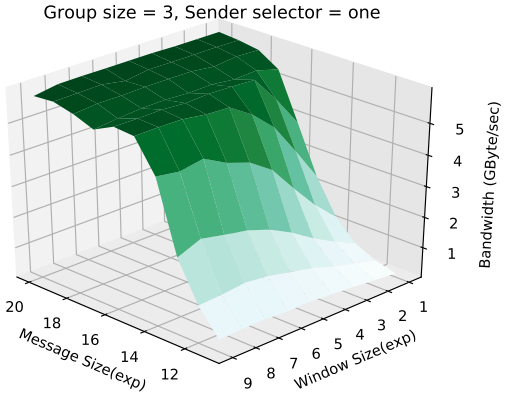
Figure 1: *bandwidth_test* throughput vs TX/RX depth and window size

We can find a same trend in all scenarios: the throughput grows with *window_size* when the *window_size* is small; then it declines when *window_size* is beyond some point. However, the peak point is quiet different depending on factors including message size and sender selector, probably also group size and hardware spec. This experiment will give us an idea of the trend. But in real deployment, we might need a profiling tool to find the optimal values for a given setup.

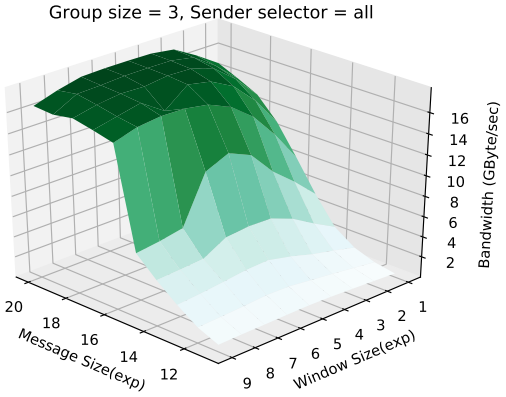
3 GPU Direct Performance

GPUDirect allows data transfer between local/remote CPU/GPU Memory using RDMA [4]. In this section, we evaluated the performance of GPUDirect using a two-node setup. Both of the nodes *A* and *B* have a 100 Gbps Mellanox ConnectX-4 card and an nVIDIA Tesla T4 card attached to NUMA node 1. Node *B* has an extra nVIDIA Tesla T4 card attached to NUMA node 0. We collected the data for the following:

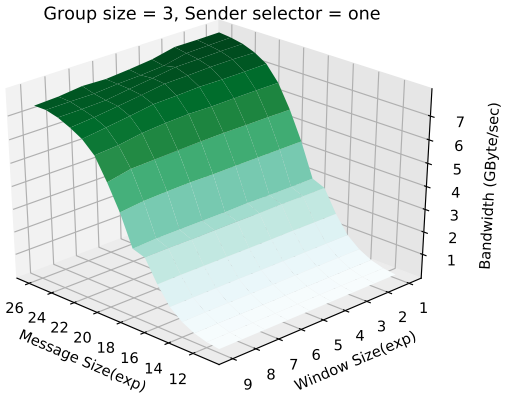
1. the throughput and latency of RDMA read/write from local CPU memory to remote CPU memory; and
2. the throughput and latency of RDMA read/write from local CPU memory to remote GPU memory; and
3. the throughput and latency of RDMA read/write from local GPU memory to remote CPU memory; and
4. the throughput and latency of RDMA read/write from local GPU memory to remote GPU memory.



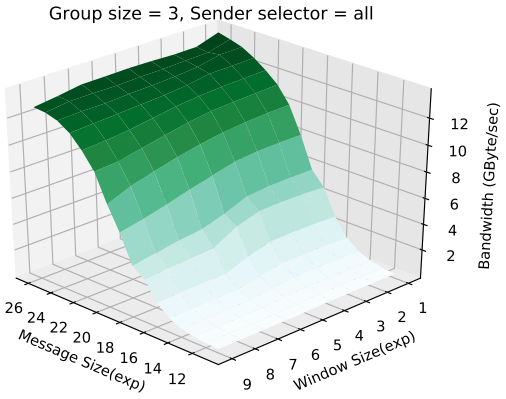
(a) SMC, One Sender



(b) SMC, All Sender



(c) RDMC, One Sender



(d) RDMC, All Sender

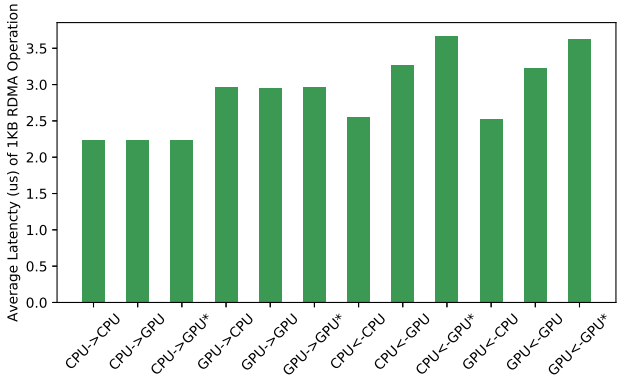
Figure 2: *bandwidth_test* throughput vs message and window size

	CPU $\rightarrow$ GPU	CPU $\leftarrow$ GPU
Same NUMA node	6954.43 MiB/s	4705.15 MiB/s
Separate NUMA nodes	6809.93 MiB/s	4454.4 MiB/s

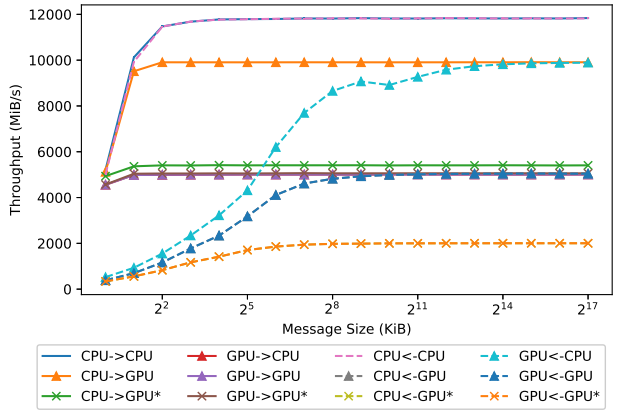
Table 3: *cuMemcpyXtoY*() Throughput Between 128 MiB Buffers

The results are shown in figure 3. Left arrow ($\leftarrow$) represents RDMA read from remote memory to local memory and right arrow($\rightarrow$) represents RDMA write from local memory to remote memory. Star ($*$) means the remote memory and Mellanox NIC are attached to different NUMA nodes. As shown in figure 3a, writing to GPU memory has a lower latency compare to reading from GPU memory. And reading from the GPU memory on another NUMA node introduces significant latency. Figure 3b shows that NUMA node locality affect throughput performance more. The throughput of RDMA write from local CPU memory to remote GPU memory reduce by $\sim 50\%$ when the remote GPU and NIC are on different NUMA nodes (See $CPU \rightarrow GPU$ vs $CPU \rightarrow GPU*$). RDMA read performance decreases more in such a compairson (See $CPU \leftarrow GPU$ vs $CPU \leftarrow GPU*$). From the same figure, we also see that reading from GPU memory is slower than writing to it. The results agree with the data in a previous benchmarking [1].

We also tested the performance of *cuMemcpyDtoH*() and *cuMemcpyHtoD*(), which are standard memory copy API from nVIDIA library. The result is shown in table 3. From the above numbers we can conclude that, compared to CPU-to-CPU RDMA plus data copy from CPU to GPU with



(a) Latency for 1 KiB Messages



(b) Throughput: $GPU \rightarrow CPU$, $GPU \rightarrow GPU$, and $GPU \rightarrow GPU*$ overlap; $CPU \leftarrow GPU$ and $GPU \leftarrow GPU$ overlap; $CPU \leftarrow GPU*$ and $CPU \leftarrow GPU*$ overlap.

Figure 3: GPUDirect Performance

cuMemcpyXXX(), GPUDirect is several times faster. Also, GPUDirect saves memory space as well as CPU cycles.

4 Image Pipeline Performance

We evaluate the performance of the Cascade image pipeline in this section. We start with a simple one-stage image pipeline using three nodes, including a computation node, a storage node, and an external client node. The cascade service consists of a computation subgroup and a storage subgroup. Each subgroup has one shard, and each shard contains one node. The computation subgroup maintains a volatile key-value store. It receives the put requests from the external client, which uploads image frames for classification. On receiving a put request, the computation node triggers the backend classification model to tell the flower name in the uploaded flower image. Then the result is written to the storage node and persistent in the file system.

Each of the three servers has two Intel Xeon Gold 6242

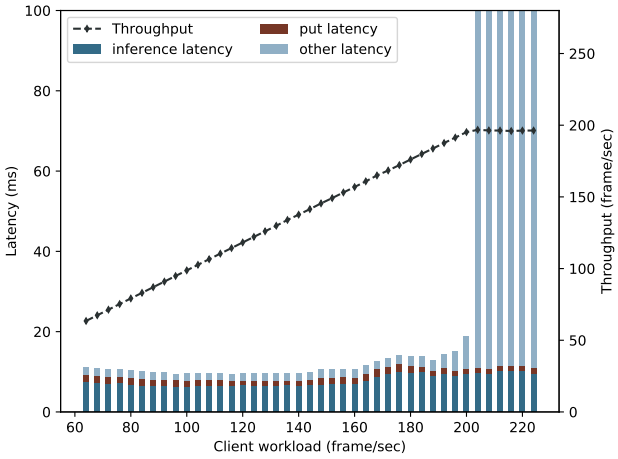


Figure 4: Performance Breakdown with 4 Workers on GPU

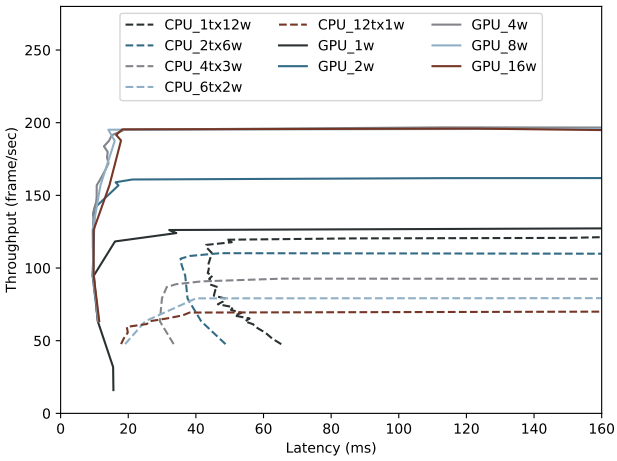


Figure 5: Throughput vs Latency

processors and 192 GBytes of memory, connected with a 100 Gbit/s InfiniBand network. The computation node runs on a server with an NVIDIA Tesla T4 GPU.

We control the frame rate at the external client side to see how the throughput and latency change. To measure the end-to-end latency, we start a UDP server on the external client node to collect the timestamp when a result of a frame is sent to the storage tier successfully. The computation node reports the timestamp to the UDP server when the storage subgroup acknowledges the corresponding write request[1]. The throughput is the number of image frames divided by the timespan from sending the first message to the result of the last image frame written.

Figure 4 shows the throughput and latency breakdown of the image pipeline, where four classification worker threads

[1]Please note that timestamp indicates that the result message is *STABLE* instead of *PERSISTED*.

run in the GPU. The throughput grows along with the client frame rate until the GPU gets saturated at ∼200 frames per second. Before the GPU gets saturated, the end-to-end latency is pretty stable at ∼10ms, with ∼7ms for inference, ∼ 1.5ms for writing to the storage layer shown as put latency, and ∼ 1.5ms for other overhead. When the processing power gets saturated, the latency arises abruptly because the queueing time goes up. The inference and put latency are not affected by workload pressure.

We found that the number of worker threads impacts the performance significantly. Therefore we vary that parameter and compared the performance with inference in CPU and GPU. For CPU inference, we use 12 CPU cores and change the number of workers and the number of threads in each worker, shown as `w` and `t` respectively in figure 5. The common trend for all series is, as we saw in Figure 4, the latency keeps stably low until the computation resources are fully utilized, and grows abruptly after that. We need multiple workers to leverage the parallelism in GPU. In this particular workload, we need at least four workers. Inference with CPU is interesting: if a worker uses more threads, it achieves shorter latency but lower throughput per CPU core. A worker using 12 threads with 12 CPU cores can achieve a latency lower than 20 ms comparing to the ∼50 ms latency with one thread per worker. However, 12 one-thread workers can achieve twice the throughput of a worker with 12 threads. We argue that the thread scheduler should be adaptive with inference using CPU.

References

[1] Benchmarking gpudirect rdma on modern server platforms. `https://developer.nvidia.com/blog/benchmarking-gpudirect-rdma-on-modern-server-platforms/`.

[2] Cascade github project. `https://github.com/Derecho-Project/cascade`.

[3] Derecho github project. `https://github.com/Derecho-Project/derecho`.

[4] Gpudirect. `https://developer.nvidia.com/gpudirect`.