2022 APAC HPC-AI
Communications Performance with UCX: Dataframe Merging

Team NTHU-1

October 14, 2022

In this task, we are to run a dataframe merging benchmark on Gadi nodes, which communicates with each other via UCX *(Unified Communication X)*.

1 Optimized Configurations

We tried various sorts of UCX options. Most of them were `export`ed as environment variables in the `cluster.cfg`.

1.1 Ensure CUDA device

We've noticed that there is a part of code which creates a local UCX network in file `utils_multi_node.py`. There are several modifiable parameters, and one of them is `ensure_cuda_device`.

According to the comments in `utils_multi_node.py`, setting `ensure_cuda_device` to `True` sets the `CUDA_VISIBLE_DEVICES` environment variable to match the proper CUDA device based on the worker's rank. Also, having this set to `False` may cause all workers to use device 0, thus potentially leading to low performance. Hence we set `ensure_cuda_device` to `True`, which improves the performance from 3.95 GiB/s to 4.27 GiB/s (108.1% speedup). We continued the following experiments with this modification.

1.2 Enable *Hardware Tag Matching*

In *Tag Matching*, the software holds a list of matching entries called matching list. Each matching entry contains a tag and a pointer to an application buffer. The matching list is used to steer arriving messages to a specific buffer according to the message tag. The action of traversing the matching list and finding the matching entry is called *Tag Matching*.

Sending messages with numeric tags accelerates the processing of incoming messages, leading to better CPU utilization and lower latency for expected messages. Currently, *Hardware Tag Matching* is supported for the accelerated RC *(Reliable Connected)* and DC *(Dynamic Connected)* transports, and we found that it can be enabled by setting the following environment parameters:

- `UCX_RC_MLX5_TM_ENABLE=y`
- `UCX_DC_MLX5_TM_ENABLE=y`

By setting these two environment parameters, the throughput improved from 4.27 GiB/s to 4.36 GiB/s (102.1% speedup).

1.3 Use mutex for Multithreading Support

The environment variable `UCX_USE_MT_MUTEX` is set to `n` by default, which means not using **mutex** for multithreading support and using **spinlock** instead.

Both **spinlock** and **mutex** are common synchronization mechanism. The difference between them is: the mechanism that **mutex** uses is sleep-waiting, while the mechanism that **spinlock** uses is busy-waiting. That is, when a thread tries to lock a **mutex** and it does not succeed, it will go to sleep, immediately allowing another thread to run. As for **spinlock**, as long as the **spinlock** polling is blocking the only available CPU core, no other thread can run and the lock won't be unlocked either.

There are some pros and cons for both of them, listed below:

Pro for mutex Having better adaptability.

Con for mutex Costs more resources since require switches.

Pro for spinlock Costs less resources, compared to mutex.

Con for spinlock Spend more user time since threads spend time on waiting at its core.

Since we have no idea which one is more efficient for the task, we've tried both of them and gained the following results:

Throughput with spinlock 4.27 GiB/s

Throughput with mutex 4.71 GiB/s

We've found out that we should choose **mutex** instead of **spinlock**, i.e., having `UCX_USE_MT_MUTEX` set to `y`, to get better performance, the throughput increased from 4.27 GiB/s to 4.71 GiB/s (110.3% speedup).

1.4 Enable Non-blocking Mode

The option `UCXPY_NON_BLOCKING_MODE` determines whether to use non-blocking mode to recieve and send data. We discovered that the performance improves if we enable non-blocking mode, i.e, setting `UCXPY_NON_BLOCKING_MODE` to 1: the throughput increased from 4.27 GiB/s to 4.96 GiB/s (116.1% speedup).

1.5 Switch *Rendezvous* protocol to *Active Messages* scheme

There is a environment parameters that manipulate communication scheme in *Rendezvous* protocol. Although there are only 3 options: `get_zcopy`, `put_zcopy`, `auto`, documented in detail, we found that there are several other schemes such as `get_ppln`, `put_ppln` (these two might refer to pipe-lined get / put), `am`, `rkey_ptr`; among all these schemes `am`, which stands for *Active Messages*, worked better than any other scheme above.

After setting `UCX_RNDV_SCHEME=am`, the throughput increased from 4.27 GiB/s to 5.54 GiB/s (129.7% speedup), which is a significant improvement.

1.6 Disable Nagle algorithm

By setting the option `UCX_TCP_NODELAY` to `y`, we can set `TCP_NODELAY` socket option to disable Nagle algorithm, which provides better performance: the throughput increased from 4.27 GiB/s to 4.40 GiB/s (103.0% speedup) after we exporting this environment parameter.

1.7 Other parameters

1.7.1 Adjust the initial pool size for memories

In the dask training, the lecturer mentioned that we can adjust the `--rmm_pool_size` parameter to determine the size of memory to allocate at the beginning while initializing a cluster. RMM is memory manager from RAPIDS. Since memory allocation is an expensive operation, allocating a big chunk of memory improves performance.

Similarly, we found the parameter `--rmm-init-pool-size` in file `run-cluster.sh`, and we thought about that setting the initial memory size which exactly same as usage may reduce waste of memory and maybe improve the performance.

Without any optimal configuration, the performance do increase if we make the initial pool size smaller. The default `--rmm-init-pool-size` is set to 3×10^{10}. If we set to 2.5×10^{10}, the throughput increased from 4.27 GiB/s to 4.74 GiB/s; if setting to 2×10^{10}, the throughput increased from 4.27 GiB/s to 4.35 GiB/s (101.9% speedup).

However, with the optimal configurations, setting `--rmm-init-pool-size` smaller doesn't guaranteed that the performance would be better. The best throughput we could achieve with `--rmm-init-pool-size` set to 3×10^{10} is 8.98 GiB/s. If we set it to 2.5×10^{10} and 2×10^{10}, then the throughput would come to 8.25 GiB/s and 8.27 GiB/s, respectively, which is a bit worse than the best performance.

1.7.2 Enable various optimizations

Setting `UCX_UNIFIED_MODE` to `y` enables various optimizations intended for homogeneous environment. According to the document, enabling this mode implies that the local transport resources/devices of all entities which connect to each other are the same. The throughput comes from 4.27 GiB/s to 4.39 GiB/s (102.8% speedup) if we enable this mode. Although it brought improvement to the performance, this option would conflict to `UCX_RNDV_SCHEME=am`. Since the latter option could bring better performance to the task, we decided not to enable this mode.

1.7.3 Modify the buffer grow rate

There are two options that determine how much buffers are added every time the memory pool grows. `UCX_TCP_RX_BUFS_GROW` determines for the receive memory pool, while `UCX_TCP_TX_BUFS_GROW` determines for the send memory pool. We wonder that whether modifying the buffer grow rate could bring better performance for us. Therefore, we've tried to increase the buffer grow rate. The default value of `UCX_TCP_RX_BUFS_GROW` and `UCX_TCP_TX_BUFS_GROW` are both 8. If we set both of them to 16, the throughput increases from 4.27 GiB/s to 4.68 GiB/s (109.6% speedup). Nevertheless, if we combine these two options with other optimal configurations,

2 Result

2.1 All optimal config

Combining the above options altogether, we have:

- `UCX_RC_MLX5_TM_ENABLE=y`
- `UCX_DC_MLX5_TM_ENABLE=y`
- `UCX_USE_MT_MUTEX=y`
- `UCXPY_NON_BLOCKING_MODE=1`
- `UCX_RNDV_SCHEME=am`
- `UCX_TCP_NODELAY=y`
- `UCX_CUDA_COPY_MAX_REG_RATIO=1.0`
- `UCX_MAX_RNDV_RAILS=1`
- `UCX_MEMTYPE_CACHE=n`

As the problem description mentioned, our objective is to achieve best bandwidth and throughput performance. We need to run 100 iterations, running on 16 GPUs of 4 nodes, and each 10^6 and 2.5×10^7 rows per chunk, for small data set and large data set, respectively. By modifying the parameters above, we achieve the following results:

Table 1: Average Throughput of 100 times (With Ordinary Volta GPUs)

Chunk size	**Default**	**Optimized Configuration**
10^6	3.95 GiB/s	9.28 GiB/s
2.5×10^7	4.40 GiB/s	12.37 GiB/s

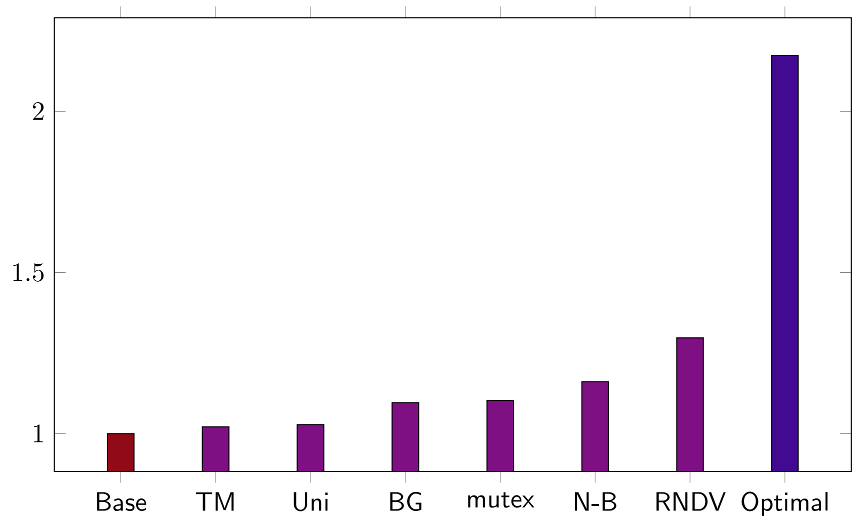

Figure 1: Bar graph of throughput speedup

Figure 1 illustrate the bar graph of throughput speedup of different options. Base, TM, Uni, Uni, BG, mutex, N-B, RNDV and Optimal represent baseline, *Hardware Tag Matching*, unified mode, buffer grow rates, non-blocking mode, *Rendezvous* scheme and the optimal combination respectively.

3 Result running on DGX-A100 nodes

We are told that there're two DGX-A100 servers in NCI Gadi, where we may run dataframe merging better, and we may earn some bonus points with the result of running on DGX-A100 nodes. We are curious about the performance with DGX-A100 nodes, thus we also experimented with the DGX servers, the results are listed below:

It's worth noting that although we mentioned that setting `UCX_RNDV_SCHEME` to `am` could optimize the performance, this option somehow doesn't work well with DGX A100 servers. Switching this parameter to `Active Messages` makes the performance worse in this case, making the throughput drop drastically from

Table 2: Average Throughput of 100 times(With DGX-A100s)

Chunk size	**Default**	**Optimized Configuration**
10^6	16.66 GiB/s	16.69 GiB/s
2.5×10^7	34.89 GiB/s	89.00 GiB/s

88.29 GiB/s to 34.89 GiB/s. On the contrary, switch back to default scheme makes the performance a bit better, which makes the throughput increases to 89.00 GiB/s.