

ECS 289D Project Report: RDMA Tutorial in the GPU Era

Yanfeng Ma, Qianqian Tan, Kaiyue Li, Zhuoli Huang, Yiqiao Lin

December 7, 2025

Abstract

We present an RDMA programming tutorial that targets readers who are new to RDMA. The tutorial covers basic verbs concepts (Queue Pairs, Completion Queues, memory registration), the idea of GPU-Direct RDMA, and provides a sequence of code examples that progressively introduce RDMA read/write examples, one-sided vs two-sided operations, reliable connection (RC) vs unreliable datagram (UD), as well as more advanced topics such as RDMA optimizations in UCCL, and EFA NIC on AWS.

The tutorial is available as a public website at <https://kyli-leo.github.io/289D-RDMA-tutorial/>, and all code and scripts are hosted on GitHub at <https://github.com/kyli-Leo/289D-RDMA-tutorial>.

1 Introduction

Remote Direct Memory Access (RDMA) is a key building block for high-performance networking in modern datacenters, especially in this LLM Era to facilitate GPU computation. However, the RDMA contains a complex set of APIs and many settings. While there are RDMA examples available online, they are usually not documented well or lack of coding examples.

In this course project, we design and implement a web-based RDMA tutorial that is customized for beginners to understand RDMA.

Our tutorial focuses on the following objectives:

- Providing a clear and in-depth introduction to RDMA, including its core concepts and why it plays a critical role in modern datacenters.
- Building up from fundamental RDMA verbs to queue pairs, and ultimately to the principles and mechanisms of memory registration.
- Delivering a minimal, self-contained collection of C/C++ examples that showcase the essential code required to run RDMA in a client-server environment.
- Supplying reproducible benchmarking scripts capable of demonstrating millions of operations per second and scalable bandwidth behavior across message sizes.
- Explaining advanced RDMA programming optimizations, with particular emphasis on techniques used in the UCCL framework.
- Introducing specialized RDMA-like technologies, such as Amazon's Elastic Fabric Adapter (EFA), and discussing how they relate to conventional RDMA.

Our tutorial website is hosted at <https://kyli-leo.github.io/289D-RDMA-tutorial/>, and the corresponding source code is publicly available at <https://github.com/kyli-Leo/289D-RDMA-tutorial>.

2 Background and Motivation

Remote Direct Memory Access (RDMA) allows one machine to directly read or write another machine's memory without involving the remote CPU or operating system. Its three core attributes—asynchronous queues (high concurrency with no blocking), kernel bypass (avoiding syscalls and context switches), and one-sided operations (hardware-driven remote memory access)—enable ultra-low latency, low CPU overhead, zero-copy transfer, and high throughput[1]. These properties make RDMA widely used in data centers, HPC, and distributed AI training.

RDMA's basic abstractions include Queue Pairs (QPs) consisting of send and receive queues(SQ,RQ). Applications post work requests (WRs) to these queues, the NIC executes them, and results are delivered through Completion Queues (CQs)[3]. QPs operate within a Protection Domain (PD) and on registered Memory Regions (MRs). RDMA supports several transport types: RC (reliable, supporting send/recv, read/write, and atomics), UC (unreliable, supporting send/recv and RDMA write), and UD (best extendibility , unreliable, supporting only send/recv)[2].

RDMA provides two-sided and one-sided operations. Two-sided verbs (send/recv) require both sides to pre-post matching WRs, making them useful for control paths and connection setup. One-sided operations (read/write/atomic) directly access remote memory without remote CPU involvement, making them ideal for key-value stores, distributed shared memory, and high-throughput data paths.

Despite RDMA's performance advantages, its programming model is challenging: developers must manage low-level verbs, configure QPs and CQs, register memory correctly, and understand transport semantics that differ significantly from traditional sockets. Many examples also assume specific cluster environments and lack clear guidance. Therefore, a practical RDMA tutorial is valuable for introducing the verbs API and providing real, cluster-adapted examples to help users learn and deploy RDMA.

3 Tutorial Design

3.1 Website Structure

The website is structured into several components:

- **Homepage.** Provides a high-level overview of the tutorial's goals, the course context, and guidance on how readers should navigate the material.
- **RDMA Basics.** This section introduces fundamental concepts of RDMA programming and is organized into the following subsections:
 - *Introduction to RDMA:* an overview of why RDMA matters in modern high-performance systems.
 - *RDMA Verbs API:* explanation of the verbs programming model and the roles of send, receive, write, and read operations.
 - *Memory Registration:* discussion of memory regions, keys, and the necessity of registration for both one-sided and two-sided operations.
 - *Queue Pairs:* description of queue pair states, connection setup, and how work requests traverse the queues.
- **GPUDirect RDMA.** Describes how RDMA interacts with GPU memory, including setup procedures, constraints, and differences relative to CPU-memory RDMA.
- **Code Examples.** This part contains the minimal, self-contained examples referenced by the text:

- *RDMA Read/Write Example*: a baseline client–server program demonstrating read/write in CPU RAM.
- *RDMA Read/Write Example with GPU-Direct (ROCm)*: a baseline client–server program demonstrating read/write in GPU memory
- *One-sided vs Two-sided Test*: benchmarks comparing send/receive with write under different message sizes. The tests compare the CPU and GPU ram performance.
- *RC vs UD Test*: experiments contrasting reliable connection (RC) and unreliable datagram (UD) modes.
- *UCCL Optimizations*: discussion of advanced RDMA optimizations introduced in the UCCL framework.
- *AWS EFA Notes*: brief remarks on RDMA-like features on the AWS Elastic Fabric Adapter (EFA) and their differences from standard InfiniBand/RoCE environments.

Overall, the website structure mirrors the intended learning pathway: readers begin with core RDMA concepts, progress to GPU-related extensions, and finally explore practical examples and reproducible performance benchmarks.

3.2 Design goals

- Minimal code for readers to understand RDMA

Rather than implementing the full RDMA state machine from scratch, we leverage `rdma_cm` to abstract away connection establishment and Queue Pair negotiation.

- Easy to reproduce results

For many of our tests, there are dataset and scripts that can semi-automate the experiments.

- Understandable concepts and data visualization.

Clear figures and diagrams explaining the results of experiments for the ease of understanding.

4 Implementation

4.1 Code Organization

Our actual code is distributed in the following layout

- `rdma_tutorial/docs/code_examples/code/basic_read`: RDMA basic example for read with CPU RAM
- `rdma_tutorial/docs/code_examples/code/basic_write`: RDMA basic example for write with CPU RAM
- `rdma_tutorial/docs/code_examples/code/basic_read_gpu`: RDMA basic example for read with GPU RAM
- `rdma_tutorial/docs/code_examples/code/basic_write_gpu`: RDMA basic example for write with GPU RAM
- `rdma_tutorial/docs/code_examples/code/one_side_vs_two_side`: RDMA write vs send test (both CPU and GPU)
- `rdma_tutorial/docs/code_examples/code/RC_vs_UD`: RDMA RC vs UD test

4.2 RDMA Examples

We provide four minimal examples demonstrating one-sided RDMA operations over both CPU and GPU memory: (1) CPU RDMA Write, (2) CPU RDMA Read, (3) GPU-direct RDMA Write, and (4) GPU-direct RDMA Read.

Verbs Used. Each program issues exactly one one-sided operation:

- **RDMA Write** (`IBV_WR_RDMA_WRITE`) — the client pushes data into a remote buffer.
- **RDMA Read** (`IBV_WR_RDMA_READ`) — the client pulls data from a remote buffer.

Connection Establishment. All examples use `librdmacm` for address resolution, route resolution, QP creation, and for passing a small metadata struct (remote address, RKey, length) through the private-data field during `rdma_connect()` / `rdma_accept()`. This metadata exchange is sufficient for issuing Read/Write without any two-sided communication.

Memory Registration. CPU versions allocate host memory using `aligned_alloc()`, while GPU versions allocate device memory using `hipMalloc()`. Both are registered with `ibv_reg_mr()` in the same way, allowing the NIC to perform DMA directly into or out of the registered buffer.

All four examples share the same minimal RDMA workflow:

connection setup → memory registration → private-data exchange → one-sided operation → completion.

5 Evaluation

5.1 Experimental Setup

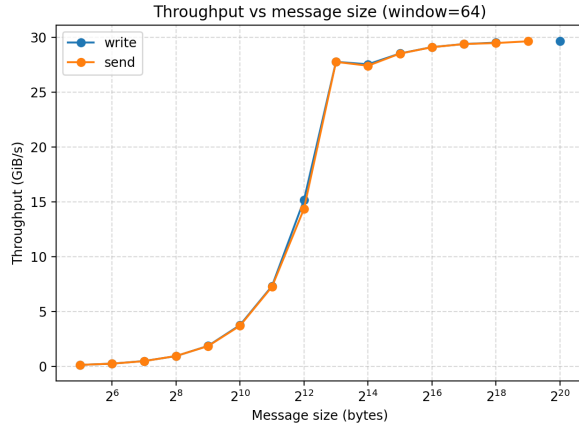
Our experiments run on two nodes in a cluster that equipped with AMD MI325x GPUs and Broadcom 400 Gb/s Ethernet NICs (RoCEv2).

5.2 Experiment results

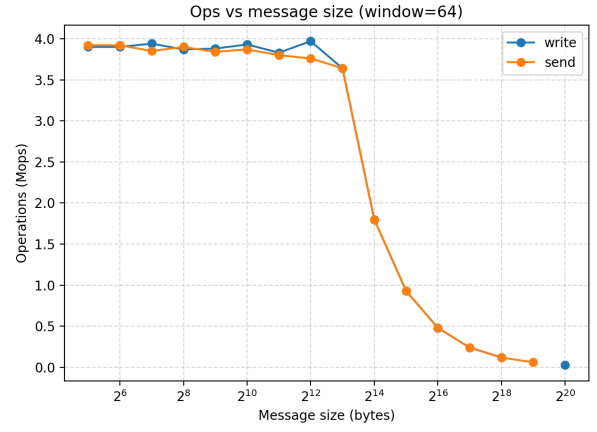
5.2.1 One-sided vs two-sided (write vs send)

We set the queue depth to be 64 and 256, then test the result of write and send in both settings. We change the message size and measure the MOPS (millions of operations per second) and Bandwidth (Gib/s).

The following are the results in plot:

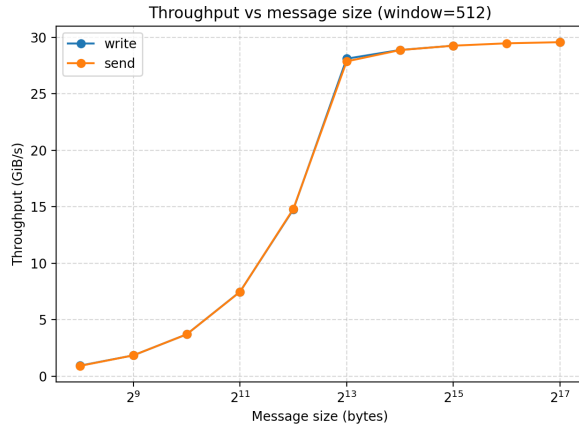


(a) Throughput (GiB/s)

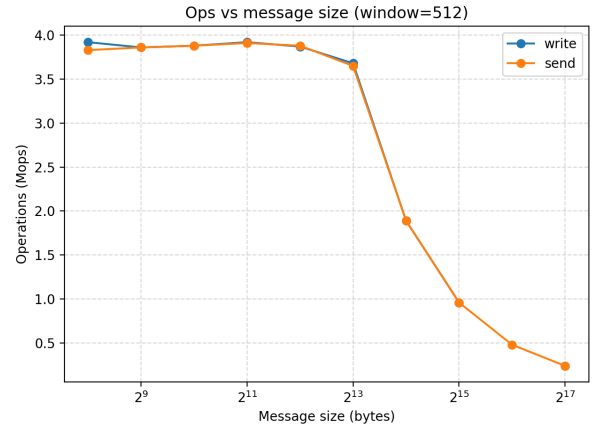


(b) MOPS

Figure 1: CPU benchmark with window = 64

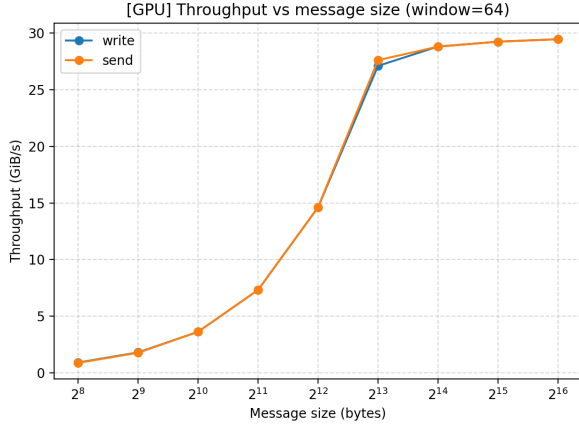


(a) Throughput (GiB/s)

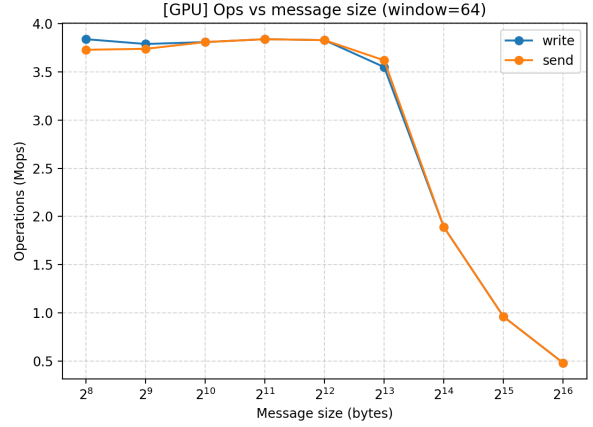


(b) MOPS

Figure 2: CPU benchmark with window = 512

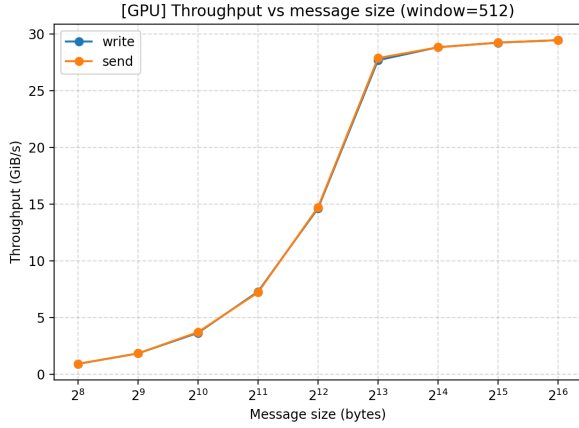


(a) Throughput (GiB/s)

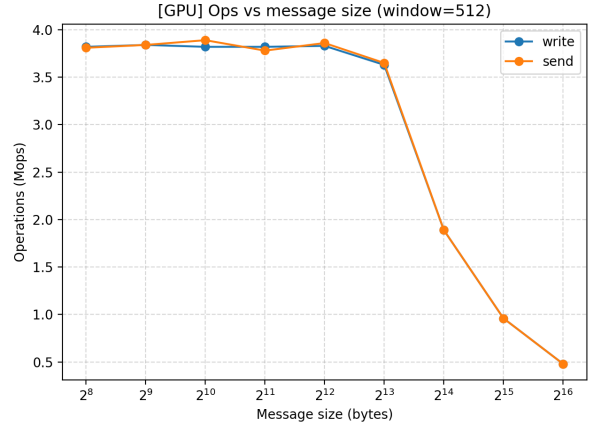


(b) MOPS

Figure 3: GPU benchmark with window = 64



(a) Throughput (GiB/s)



(b) MOPS

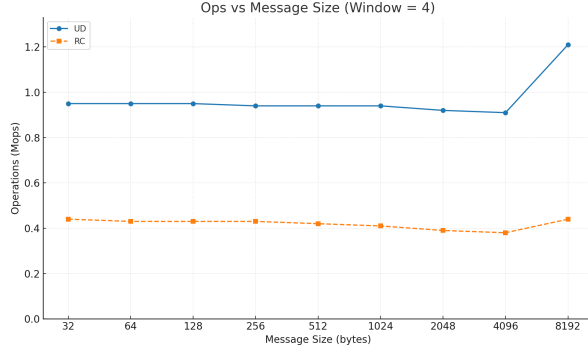
Figure 4: GPU benchmark with window = 512

We can reach a counter-intuitive conclusion from the previous results. With either a window size of 64 or 256, the send and write operations show no statistically significant performance difference in our experiment setting. An interesting observation is that at the message size 2^{13} byte or 8KB, the throughput reach its peak while maintaining the peak MOPS. This could be a relatively good message size to pick if we want to minimize the message size (reduce latency) while ensure high throughput.

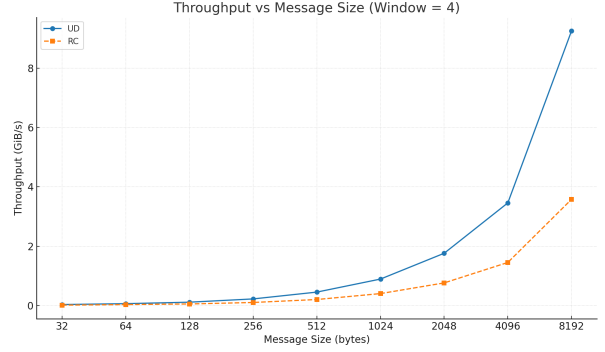
5.2.2 RC vs UD

We compare the performance difference between RC (Reliable Connected) and UD (Unreliable Datagram) under various message sizes and queue depths. In particular, we sweep message sizes from 32 B to 8192 B, and test with window sizes of 4 and 64 to observe shallow versus deep pipeline behavior. For each configuration we measure both MOPS (millions of operations per second) and throughput in GiB/s.

The following figures summarize our results:

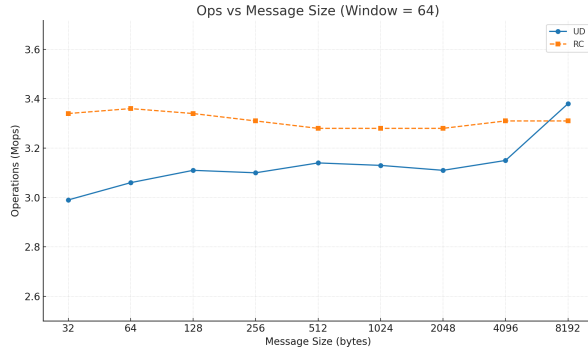


(a) MOPS (Window = 4)

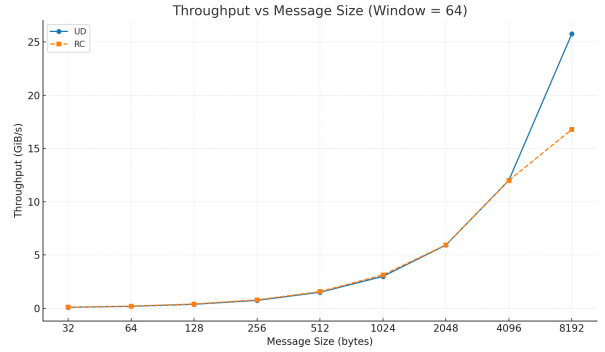


(b) Throughput (GiB/s)

Figure 5: RC vs UD benchmark with window = 4



(a) MOPS (Window = 64)



(b) Throughput (GiB/s)

Figure 6: RC vs UD benchmark with window = 64

From the above results, we observe several clear trends. With a shallow window size of 4, UD consistently outperforms RC in both MOPS and throughput. This is expected because RC must handle reliability and ordering overhead (ACKs, retransmissions, and in-order guarantees), which cannot be hidden when only a few work requests are in flight.

When the window deepens to 64, the overhead of RC becomes largely amortized by pipelining. Under this setting, RC and UD achieve nearly identical MOPS for small message sizes, indicating that the reliability cost can be effectively hidden. However, for larger messages (4 KB and above), UD still scales better and maintains higher throughput due to its lighter-weight protocol and lower per-packet overhead.

Overall, UD provides the best raw performance, especially at small window sizes or with large messages. While RC offers reliability at the cost of higher protocol overhead. A sufficiently deep pipeline allows RC to approach UD’s performance, but UD remains advantageous in high-throughput scenarios.

6 Related Work

The RDMA tutorial slides [1] offer a clear introduction to the RDMA programming model, and the survey [2] summarizes transport types and major optimization trends. However, neither resource sufficiently addresses the needs of accelerator-centric systems. Existing tutorials are largely CPU-oriented and provide limited guidance on GPU memory registration, data movement, and completion handling in GPU RDMA

workflows. Likewise, surveys emphasize high-level abstractions rather than concrete, reproducible implementations. As a result, crucial practical details are scattered across vendor documents that often focus on configuration over pedagogy and rarely present coherent end-to-end examples.

We also examined GPU-related RDMA techniques, such as NVIDIA’s GPUDirect RDMA documentation [4], which describes how to enable direct network access to GPU memory. However, these documents are lengthy and often lack clear, instructive examples. To obtain a more accessible entry point, we referred to RDMA tutorials such as Rhiswell’s guide [5] and Insu Jang’s introduction to InfiniBand and RDMA [3], both of which provide practical explanations of RDMA primitives and workflows.

Building on these resources, we aim to develop a new tutorial that consolidates our understanding of RDMA and introduces GPU-oriented optimizations inspired by UCCL’s design.

7 Individual Contributions

[Kaiyue Li] Design the tutorial structure and set up the website with GitHub actions deployment. Write the GPU-Direct chapter, RDMA read examples. Perform experiments regarding write vs send and the effect of message size. Generate graph and write the experiment analysis. Design the structure of final report.

[Qianqian Tan] Implemented the RDMA write examples. Write the EFA NIC part of the tutorial.

[Yanfeng Ma] Implemented the RDMA RC and UC test. Generate graph and write the experiment analysis.

[Yiqiao Lin] RDMA Queue Pairs introduction, Share receive queue and Share completion queue in UCCL analysis. Write the related work part of the report.

[Zhuoli Huang] RDMA Verbs API introduction. Chained post and Scatter gather list in UCCL analysis. Write the Background and Motivation part of the report

8 Future Work

Due to the time limitation, we cannot cover every aspect of RDMA in our tutorial. The following is a couple of points that could be implemented in the future.

- **Optimization-related performance analysis** All experiments in this project use a single queue pair and intentionally avoid advanced optimizations. In future work, we measure the performance of common RDMA optimizations, such as those employed in UCCL, and systematically analyze their impact on both throughput and operation rate.
- **Visualization Tools.** Since this project aims to be an educational tutorial, future work may include animations of QP states, WQE/CQE flows, and NIC pipelines to help readers better understand RDMA’s execution model.

9 Acknowledgment

Portions of the writing, experiment code, and code explanations were assisted by AI tools. All technical decisions, experiments, and validations were performed by the authors.

References

- [1] Jelena Giceva. Ssc18: Rdma tutorial. Course notes, Department of Computing, Imperial College London, 2018. Accessed: 2025-12-02.
- [2] Mandi Huang, Tao Li, Hui Yang, Chenglong Li, Yutao Zhang, and Zhigang Sun. Survey on ethernet rdma network interface card. *Journal of Computer Research and Development*, 62(5):1262–1289, 2025.
- [3] Insu Jang. Introduction to programming infiniband rdma. <https://insujang.github.io/2020-02-09/introduction-to-programming-infiniband/>, 2020. Accessed: 2025-12-02.
- [4] NVIDIA Corporation. *NVIDIA GPUDirect RDMA Documentation*, 2024. Accessed: 2025-12-02.
- [5] rhiswell. rdma-tutorial. <https://github.com/rhiswell/rdma-tutorial>, 2025. GitHub repository, commit abc (as of 2025-12-02).