Designing RDMA-based efficient Communication for GPU Remoting

Shreya A Bhandare

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Applications

Dimitrios S. Nikolopoulos, Chair
Ali R. Butt
Kirk W. Cameron

June 30, 2023
Blacksburg, Virginia

Keywords: GPGPU, Virtualization, RDMA, CUDA

Designing RDMA-based efficient Communication for GPU Remoting

Shreya A Bhandare

(ABSTRACT)

The use of General Purpose Graphics Processing Units (GPGPUs) has become crucial for accelerating high-performance applications. However, the procurement, setup, and maintenance of GPUs can be costly, and their continuous energy consumption poses additional challenges. Moreover, many applications exhibit suboptimal GPU utilization. To address these concerns, GPU virtualization techniques have been proposed. Among them, GPU Remoting stands out as a promising technology that enables applications to transparently harness the computational capabilities of GPUs remotely[7]. GVirtuS, a GPU Remoting software, facilitates transparent and hypervisor-independent access to GPGPUs within virtual machines. This research focuses on the middleware communication layer implemented in GVirtuS and presents a comprehensive redesign that leverages the power of Remote Direct Memory Access (RDMA) technology. Experimental evaluations, conducted using a matrix multiplication application, demonstrate that the newly proposed protocol achieves approximately 50% reduced execution time for data sizes ranging from 1 to 16MB, and around 12% decreased execution time for sizes ranging from 500 to upto 1GB. These findings highlight the significant performance improvements attained through the redesign of the communication layer in GVirtuS, showcasing its potential for enhancing GPU Remoting efficiency.

Designing RDMA-based efficient Communication for GPU Remoting

Shreya A Bhandare

(GENERAL AUDIENCE ABSTRACT)

General Purpose Graphics Processing Units (GPGPUs) have become essential tools for accelerating high-performance applications. However, the acquisition and maintenance of GPUs can be expensive, and their continuous energy consumption adds to the overall costs. Additionally, many applications often underutilize the full potential of GPUs. To tackle these challenges, researchers have proposed GPU virtualization techniques. One such promising approach is GPU Remoting, which enables applications to seamlessly utilize GPUs remotely. GVirtuS, a GPU Remoting software, allows virtual machines to access GPGPUs in a transparent and independent manner from the underlying system. This study focuses on enhancing the communication layer in GVirtuS, which facilitates efficient interaction between virtual machines and GPUs. By leveraging advanced technology called Remote Direct Memory Access (RDMA), we achieved significant improvements in performance. Evaluations using a matrix multiplication application showed a reduction of approximately 50% in execution time for small data sizes (1-16MB) and around 12% for larger sizes (500-800MB). These findings highlight the potential of our redesign to enhance GPU virtualization, leading to better performance and cost-efficiency in various applications.

Dedication

Dedicated to my loving family

Acknowledgments

First, I would like to thank my adviser, Dr. Dimitrios Nikolopoulos, who patiently guided me through the process of the M.S. program as well as my research work. His encouragement and understanding have not only helped me in my work but also in my job search during this difficult time. I am also grateful for his flexibility during my unforeseen personal health problems. Finally, I would like to thank Dr. Ali R Butt and Dr. Kirk Cameron for their guidance and for serving on my thesis committee.

Contents

List of Figures

List of Tables

List of Abbreviations

API Application Porgramming Interfac

CPU Central Processing Unit

CQE Completion Queue Element

GPGPU General Purpose Graphics Processing Units

GPU Graphics Processing Units

HCA Host Channel Adapter

IP Internet Protocol

NIC Network Interface Card

OS Operating System

QP Queue Pair

RDMA Remote Direct Memory Access

RoCE RDMA over Converged Ethernet

TCP Transmission Control Protocol

WQE Work Queue Element

Chapter 1

Introduction

1.1 Problem Statement

GPU remoting is a hypervisor-independent GPU virtualization technique that offers ease of implementation and transparency. It has application in low-end, edge and IoT devices as they are not equipped with high specification hardware. However, it is known to be relatively slower compared to other solutions in this space. To improve the performance of GPU remoting, it is crucial to address the communication overhead caused by redirecting APIs and data to a remote accelerator over a network. The existing approach used by GPU remoting software, such as GVirtuS[9], relies on TCP's byte-by-byte transfer, which is proven to be less efficient. Additionally, the naive sequential approach employed by GVirtuS, where APIs are sent one by one and each call waits for the result before sending the next API call to the remote GPU, contributes to further performance limitations. Therefore, there is a need for an evolution in GPU remoting techniques to enhance their performance by optimizing communication efficiency and adopting more efficient API transfer strategies.

1.2 Design & Contributions

In our research, we focus on reducing the communication overhead associated with GPU remoting, specifically targeting the GVirtuS software. Our primary contribution is the

introduction of a new communication protocol based on RDMA (Remote Direct Memory Access) over converged Ethernet. We thoroughly investigate various RDMA operations and select the channel-based RDMA Send/Recv API to reduce synchronization complexities at both the client and server sides.

To optimize the RDMA protocol for efficient communication, we carefully tune multiple parameters to ensure compatibility with this specific type of use case of GPU remoting. Furthermore, we develop a novel batching-based communication protocol that enables the concurrent execution of network transfers, API execution, and host-to-device memory transfers. This approach improves overall throughput and minimizes latency.

We provide a comprehensive overview of our implementation, detailing the design and implementation of the modified communication protocol and the specific optimizations applied to enhance the performance of GVirtuS in Chapter 4 of this thesis.

Chapter 2

Background

2.1 GPU Virtualization

GPU virtualization is a crucial process that involves abstracting and partitioning a physical GPU into multiple virtual instances, enabling efficient sharing of computational power and memory resources among multiple users or applications. This technology plays a vital role in domains such as machine learning, scientific simulations, and high-performance computing, where parallel processing capabilities are essential. Achieving GPU virtualization can be accomplished through different approaches, including hardware-based and software-based methods.

Classification of GPU virtualization techniques[11] -

1. API Remoting: GPU virtualization at a higher level in the GPU execution stack is achieved through API remoting. In this approach, a GPU wrapper library intercepts GPU calls in the guest OS and forwards them to the host OS or a remote machine with GPUs. The intercepted calls are processed remotely, and the results are returned to the guest OS. API remoting virtualizes GPUs at the library level, overcoming the limitations posed by inaccessible GPU drivers.

2. Para and Full Virtualization: GPU virtualization at the driver level in the GPU stack is accomplished through para and full virtualization. With the availability of architec-

ture documentation and reverse engineering techniques, custom GPU drivers can be developed based on this information. Para virtualization involves modifying the guest OS's custom driver to optimize sensitive operations by directly interacting with the host driver, thereby improving performance. Full virtualization, on the other hand, enables complete virtualization of GPU resources without requiring any modifications, providing emulation of GPUs.

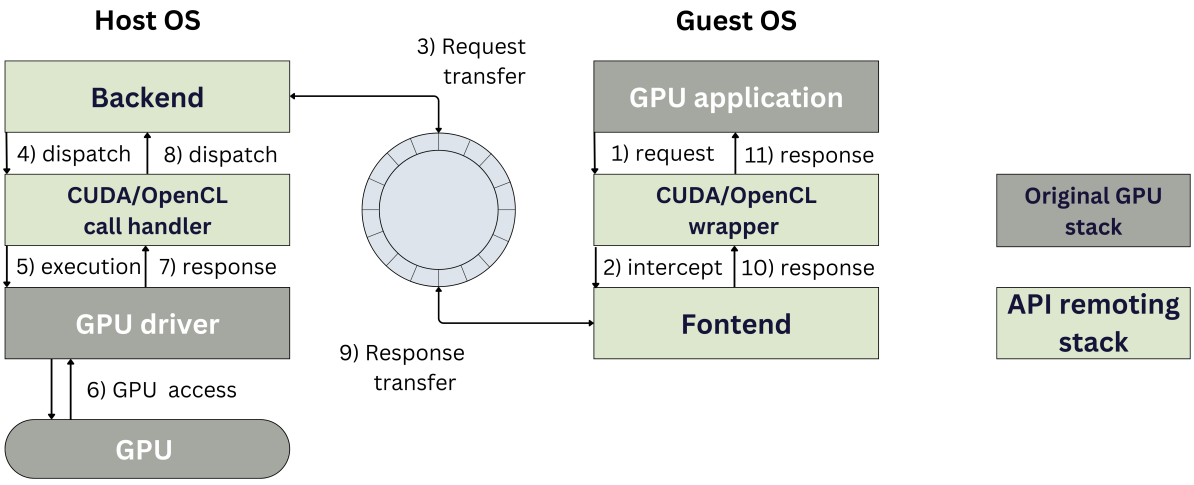
3. Hardware-Supported Virtualization: This type of virtualization leverages hardware extension features provided by chipset and GPU manufacturers to grant direct access to GPUs for a guest OS. Through the remapping of DMAs and interrupts, the chipset enables GPU pass-through access for each guest OS. However, this mechanism does not allow for the sharing of a single GPU among multiple guest OSs.

2.1.1 API Remoting

Virtualizing GPUs is rather challenging due to several reasons. First, GPU vendors do not disclose their GPU driver source code and implementation details, making it difficult to implement driver level virtualization of GPUs. Second, GPU vendors frequently introduce significant changes in each GPU generation, rendering reverse-engineered specifications obsolete. Additionally, proprietary GPU drivers provided by certain OS vendors are limited to specific operating systems, lacking universality. The absence of standard interfaces for GPU access further complicates the virtualization of these devices. API remoting has become the dominant approach for virtualizing GPUs, successfully overcoming the aforementioned challenges. This method involves providing a substitute library with an identical API to the original CUDA library in the guest OS. When applications make GPU calls, these calls are intercepted by the substitute library and redirected to the host OS, often on a remote ma-

chine equipped with GPUs. The host OS handles the calls remotely, transmitting only the results back to the application through the substitute library. By employing this technique, a virtual GPU execution environment is effectively emulated without exposing the physical GPGPU devices to the guest OS.

Figure 2.1: API Remoting Architecture and Flow



In an API remoting system, a split-device model is employed, with the frontend and backend drivers located in the guest and host OSs respectively. When a GPU call is intercepted by the wrapper library in the guest OS, it is relayed to the frontend driver. The frontend driver encapsulates the GPU operation and its parameters into a transferable message, which is then transmitted to the backend driver in the host OS. Upon receiving the message, the backend driver unpacks it and converts it back into the original GPU call. The requested operation is executed on the GPU through the GPU driver, and the resulting output is sent back to the application via the reverse path.

API remoting offers several advantages, including its compatibility with existing applications

without requiring recompilation. The dynamic linking of the wrapper library during runtime facilitates easy integration. Moreover, the straightforward architecture of API remoting minimizes virtualization overhead by bypassing the hypervisor layer. This user-space implementation also allows for independence from specific hypervisors, especially when it avoids relying on hypervisor-specific inter-VM communication methods.

However, API remoting does present challenges. One such challenge is the need to keep the wrapper libraries updated to accommodate new functions added to vendor GPU libraries. This ensures compatibility and access to the latest GPU capabilities.

Another significant challenge is the potential bottleneck in communication between the frontend and backend drivers when the GPU is located on a remote machine. The efficiency and design of the communication protocols employed are crucial in determining the application's runtime on the guest device. By optimizing the communication infrastructure, latency can be minimized, and overall application performance can be maximized. It is essential to carefully consider and optimize the communication mechanism to ensure optimal performance and user satisfaction in API remoting scenarios.

Some available API Remoting stacks are Chromium, VMGL, Blink, Parallel Desktops fro graphical applications and vCUDA, rCUDA, GVM, are Pegasus written for general purpose applications. We will be looking at GvirtuS in the next section.

2.2 GPU Execution Model

First, let's look at how CUDA programs are executed directly on a GPU before delving into GVirtuS's approach. As we know, the GPU serves as a co-processor. In this setup, the CPU handles the launch of CUDA kernels on the GPU, including data initialization and memory

transfers. To illustrate this, we examine the execution of a specific kernel. The objective of the sample CUDA kernel is to compute the addition of elements from two input arrays, a and b, and store the results in a third array, c.

To execute the provided kernel on the GPU, certain steps need to be followed. Firstly, the arrays a, b, and c must be initialized on the CPU and then transferred to the GPU. This process entails individual calls to malloc to allocate memory on the CPU, followed by initializing the arrays on the CPU. Similarly, memory allocations need to be made on the GPU as well. Subsequently, a host-to-device memory copy operation is conducted, transferring the data from the arrays on the CPU to the corresponding arrays on the GPU.

```
//Host memory allocation
double *host_a, *host_b, *host_c;
host_a = (double *)malloc(N);
host_b = (double *)malloc(N);
host_c = (double *)malloc(N);
....
....
//Device memory allocation
double *dev_a, *dev_b, *dev_c;
cudaMalloc((double **)&dev_a, N);
cudaMalloc((double **)&dev_b, N);
cudaMalloc((double **)&dev_c, N);
....
....
//Host to device data transfer
cudaMemcpy(dev_a, host_a, N, cudaMemcpyHostToDevice);
```

```
cudaMemcpy(dev_b, host_b, N, cudaMemcpyHostToDevice);
...
//CUDA Kernel
__global__ void get_addition (double *dev_a, double *dev_b, double *dev_c, int n)
{
    int g_id = blockDim.x * blockIdx.x + threadIdx.x;
    if (g_id < n) {
        dev_c[g_id] = dev_a[g_id] + dev_b[g_id];
    }
}
// Device to Host response data trasnfer
cudaMemcpy(dev_b, host_b, N, cudaMemcpyDeviceToHost);

// Free CPU arrays
...
// Free CUDA memory
...
```

These are the four types of memory allocations in CUDA:

1. Pageable Data transfer: In this type of memory allocation, data is initially copied from the host device (pageable memory) to an intermediate location known as pinned memory. From there, it is then transferred to the device memory (GPU). However, this method of transfer tends to be slow.

2. Pinned memory Transfer: With pinned memory transfer, memory is allocated and directly initialized in the host's pinned memory. By doing so, we can avoid the two

data transfers involved in pageable memory. This approach improves the overall speed of the process, although it may impact the performance of the host device.

3. Mapped memory: In mapped memory allocation, memory is directly allocated in an address space that is mapped into both the host and device memory. This allows for direct access to the memory by both the host and device. However, the transfer of data occurs during execution, which can significantly increase processing time.

4. Unified memory: Unified memory allocation creates a pool of managed memory. Each allocation from this memory pool is accessible with the same address or pointer by both the host and the device. This approach enables seamless data sharing and access between the host and device without the need for explicit memory transfers.

These different types of memory allocations in CUDA offer flexibility and trade-offs in terms of data transfer speed, performance impact, and shared memory accessibility between the host and the device.

Below are memory transfer and kernel execution times when using the different type of memories in CUDA programs. These have been calculated using the nvprof profiler for array sizes 2^{20}

memory type	memory transfer time (ms)	kernel execution time (ms)	total time (ms)
Pageable	19.43	0.44	19.87
Pinned	4.26	0.44	4.70
Mapped	0.00	3.72	7507
Unified	1.96	16.13	18.09

When comparing pinned memory to pageable memory, pinned memory has the advantage

of only requiring a single memory transfer. As a result, the time taken for memory transfer is reduced.

In the case of unified memory, the data resides in the managed pool, and transfers to the device occur on an as-needed basis. This means that the time taken for memory transfer is reduced, but the execution time for the kernel is increased. This type of memory allocation is comparable to pageable memory.

Lastly, with mapped memory, the memory is directly mapped to the device address space, eliminating the need for explicit data transfers. However, since it operates similarly to pinned memory in terms of underlying operations, the overall time required is approximately the same for both.

So, while pinned memory reduces memory transfer time, unified memory offers reduced memory transfer time at the expense of increased kernel execution time.

2.3 GVirtuS

GVirtuS is an API remoting software that uses the above mentioned 'split-driver' approach [15]. It has three main software components - the frontend, the communicator and the backend. There is another hidden component which comprises of the plugin files, which are present on both the backend and the frontend sides[16]. The primary library that we will be looking at in my work is the CUDA Runtime library, although GVirtuS has support for CuDNN, CUBLAS, CuFFT, CuRAND and others.

2.3.1 Frontend Driver

The Frontend driver is present on the client machine, where here we are assuming is not equipped with GPU acceleration capabilities. It consists of a bunch of shared object files of the CUDA library plugin, of the client side of the communicator (for eg. tcp/ip for GVirtuS) and a mediator file which packages the calls and triggers the communication operations.

The shared object files of the plugin libraries are essentially wrapper stub APIs which expose the same interface as that of the Nvidia CUDA library. When a CUDA executable (compiled with the required shared library) is run on the client device, these stub libraries intercept the CUDA calls made by the application, collects arguments, packs them into a CUDA call packet,and sends the packet to the frontend-mediator. The communication is triggered separately for every CUDA call, the mediator then waits for the response from the backend and then sends the next call. The mediator is responsible for managing the connection with the backend, sending the packets receiving the response and terminating the connection.

It is important to constantly keep updating the stub functions to match the latest version of CUDA.

2.3.2 Communicator

GVirtuS supports various runtime loadable communication modules like TCP, VMSocket, ZMQ, VMCI and Shared Memory. The communicator and it's configuration can be specified in a JSON file, along with other information required to create and manage the connection like IP address and port number. The user is mostly unaware of any implementation specifics of the protocol used, but it can vastly influence the performance of the application. GVirtuS uses a simple protocol design for all transports. The frontend sends a CUDA routine and the data that's required by the routine, packages them and sends it to the backend using a

"write" call and waits for the backend to send the response, once received we move on to the next API. This is a sequential process and hence causes delays in the processing of the application.

2.3.3 Backend Driver

The backend is essentially the server side of this setup, present on the machine that is equipped with one or more GPUs. We assume that the backend already has the CUDA runtime and other required libraries installed. Once started, it listens for connection requests from the frontend. It spawns a separate Process for every different client (frontend) to execute all requests from one client in a independent GPU context. Spawning a different server process for each remote execution over a new GPU context facilitates GPU multiplexing, also ensuring the survival of the other GPU contexts in the event of a crash of one of the server instances. Once the connection is establish with a particular frontend, the client can run as many applications as they want. The backend is compiled into an executable and runs as daemon process on the server. When it receives the CUDA calls from the frontend, it unpacks the API calls, the arguments, executes them and sends the response back to the client side.

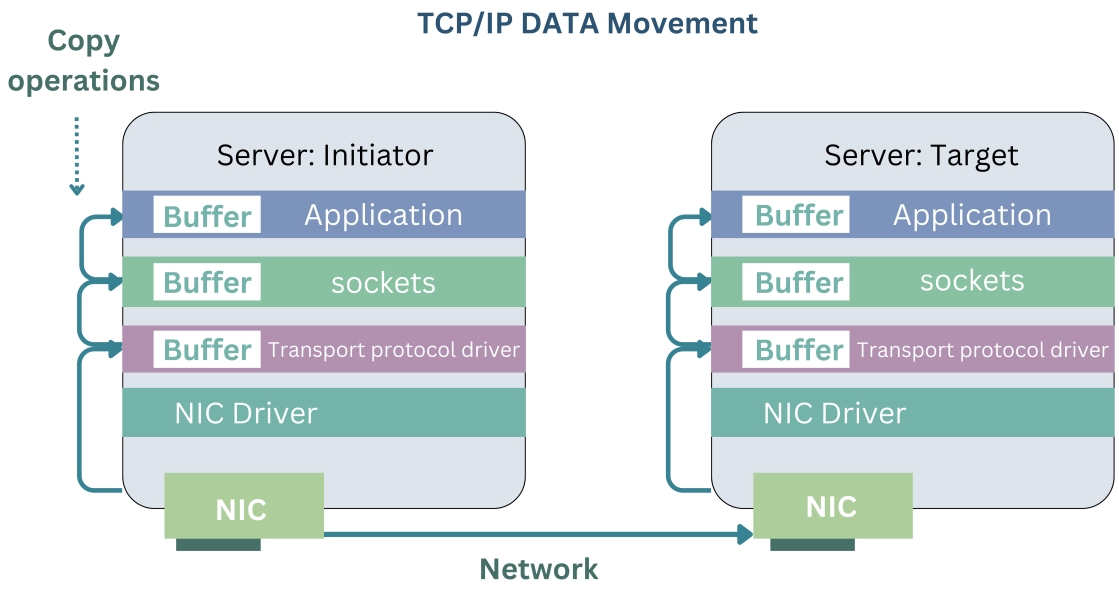
Next section takes an in-depth look at TCP/IP and why it's an unsuitable communicator for high performance workloads.

2.4 TCP/IP

TCP [2] is a reliable, connection-oriented protocol. There are various per packet overheads in the kernel that contribute to the slow speeds of TCP/IP. Every packet is copied multiple

times during the encapsulation and de-encapsulation process. The copy from user space to kernel space is a particularly expensive on. Every switch from user space to kernel space requires a context switch. These copies are resource intensive and consume multiple CPU cycles depending on the device specifications. Every packet on arriving at the NIC causes a hardware interrupt. Which again leads to a switching of modes and context switches. The hardware interrupt is dealt with in two parts, partly by the interrupt service routine and then by a soft-IRQ. Along with this kernel employs a set of data structures for incoming and outgoing packets and their metadata. sk-buff is a structure that stores per packet metadata and is allocated dynamically at runtime. There are also ring buffers for TX and RX packets as the transfer is buffered. TCP also requires CPU resources to re-assemble out of order packets increasing the overall utilization.

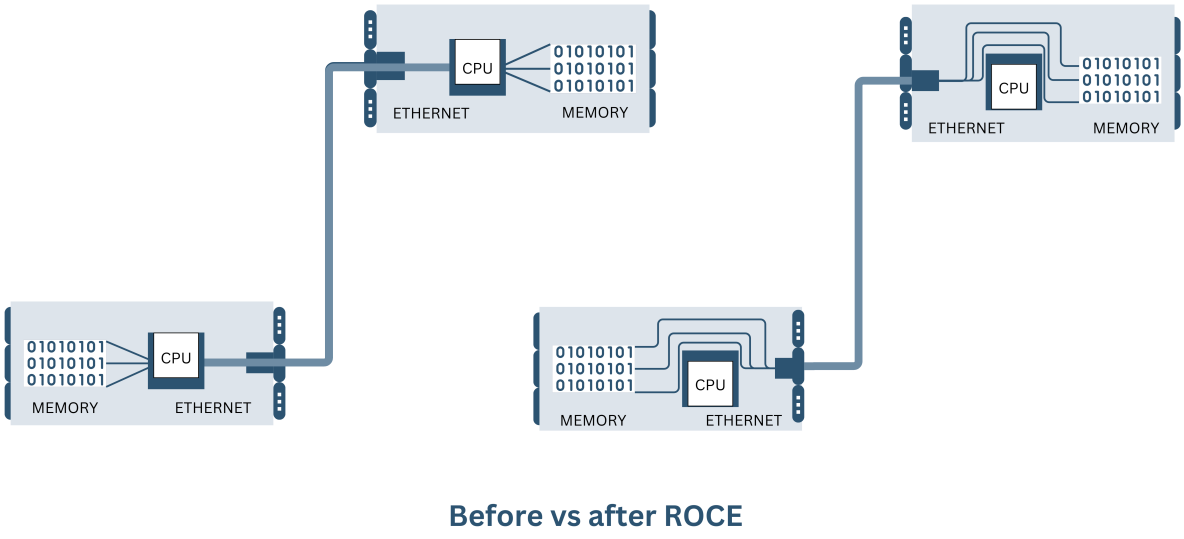
Figure 2.2: TCP/IP Data Movement



2.5 RDMA

RDMA, an extension of DMA technology, revolutionizes data access by allowing direct memory access without CPU involvement. It enables memory data to be accessed between hosts without relying on the operating system or CPU, bypassing the kernel and TCP/IP stack. By eliminating data copies between user space and kernel space and offloading the networking stack, RDMA achieves low latency and high throughput. In recent years, RDMA has gained widespread acceptance due to implementation enhancements, making it a critical technology for hyperscale data centers, particularly in high-performance computing and storage networks.

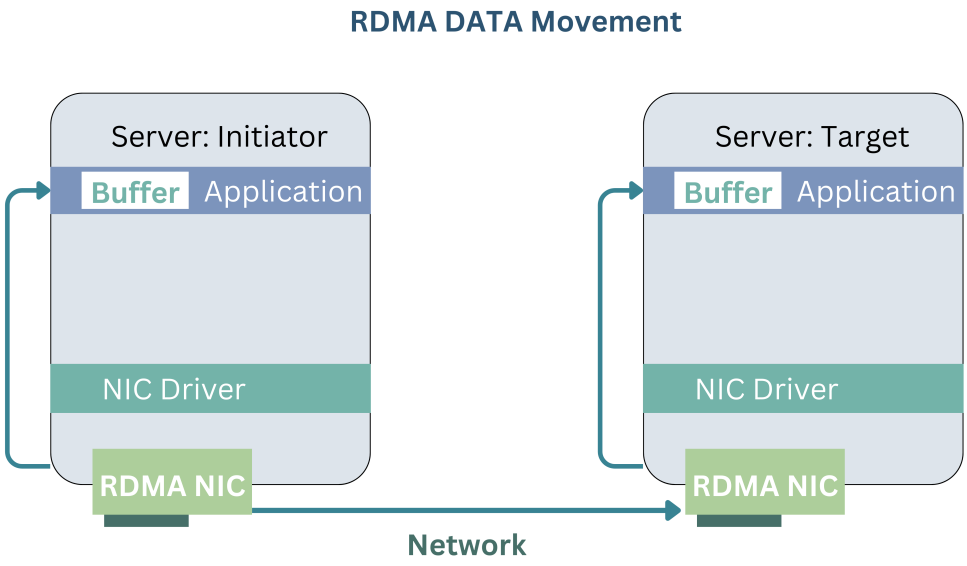
Figure 2.3: With vs Without RDMA



The RDMA protocol empowers the host adapter to determine the destination application and directly place packet contents into the application's memory space upon entering the network. This eliminates the need for CPU, cache, and context switches, enabling parallel

transfers alongside other system operations. RDMA operates on a message-based transaction model, treating data as discrete messages rather than a continuous stream. Additionally, RDMA natively supports scatter/gather entries, allowing multiple memory buffers to be read or written as a single stream.

Figure 2.4: RDMA Data Movement Flow



To harness the benefits of RDMA (Remote Direct Memory Access), a Network Interface Card (NIC) capable of RDMA, also known as an RNIC, is essential. This specialized NIC incorporates an RDMA engine called the Host Channel Adapter (HCA). The primary function of the HCA is to establish a communication channel between the RDMA engine and the application's memory. The HCA hardware plays a crucial role in enabling various RDMA protocols to be executed over the network. It contains all the necessary capabilities required for carrying out different RDMA operations efficiently. In order to facilitate data and control communication, the RDMA kernel module is employed. It establishes dedicated channels for both data and control exchange.

For the control path, the RDMA Control Manager (RDMA-CM) is utilized. This component enables the creation and management of the control channel, which is responsible for handling control operations and coordination between RDMA-enabled devices. To establish and transmit data over the data channels, the ”verbs” or ”libibverbs” API is used. This API provides a programming interface that allows applications to interact with the RDMA subsystem. Through the ”verbs” API, applications can initiate data transfers, set up memory regions for RDMA access, and manage communication endpoints.

There are multiple network protocols that support RDMA:

1. RDMA over Converged Ethernet (RoCE pronounced ’rocky’): This protocol enables RDMA transactions over an Ethernet network. It utilizes Ethernet headers for lower network layers and InfiniBand headers for upper network layers, including data. RoCE allows the use of RDMA over standard Ethernet infrastructure, and network interface cards (NICs) need to support RoCE[4].

2. InfiniBand (IB): InfiniBand is a communication link used for data flow between processors and I/O devices. It supports up to 64K addressable devices and follows the InfiniBand Architecture (IBA) standard specification. IBA defines a point-to-point switched input/output framework for interconnecting servers, communication infrastructure, storage devices, and embedded systems. InfiniBand natively supports RDMA and incorporates a lossless link layer with flow control and retransmissions to prevent congestion-based and bit error-based losses. Setting up InfiniBand requires dedicated hardware, resulting in higher initial setup costs.

3. Internet Wide Area RDMA Protocol (iWARP): iWARP allows RDMA over TCP, enabling RDMA functionality over traditional TCP/IP networks. While it supports RDMA, there are features in InfiniBand and RoCE that are not supported in iWARP.

For iWARP, NICs need to be RDMA capable and support iWARP, although software-based implementations of iWARP stacks are also possible, albeit with reduced RDMA performance advantages.

Application written with libibverbs work without requiring any modification on above mentioned network fabrics.

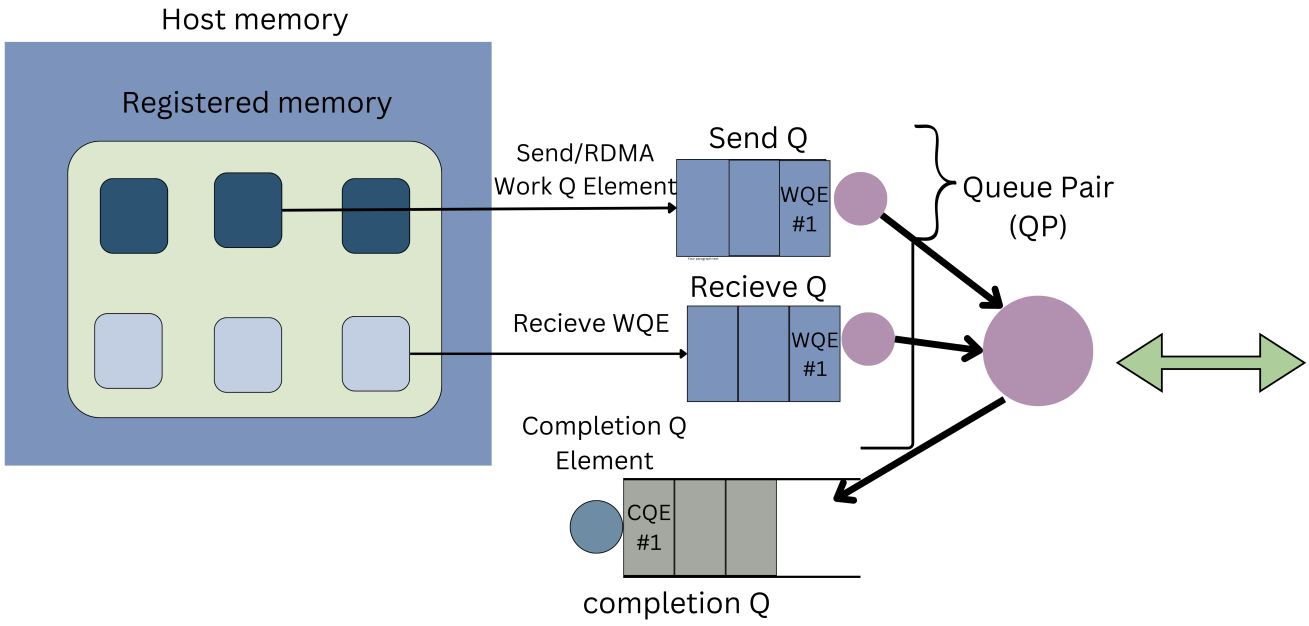
2.6 RDMA Operation

To utilize RDMA data transfer operations, memory registration is required to inform the kernel that the memory is intended for RDMA communication in a specific application. This registration process establishes a channel from the Remote Network Interface Card (RNIC) to the registered memory. It is known as memory registration. In order to enable secure access by the network device, both the source and destination buffers need to be registered for zero-copy operations. These registration and deregistration operations involve system calls and iterating over each virtual page in a manner defined by the operating system. These operations incur a significant overhead and can be more expensive than a simple memory copy, which contradicts the goal of avoiding intermediate copy overhead. To address this issue, RDMA is most practical and efficient when memory registration can be performed outside the critical path, i.e., the data path. RDMA allows registration of various types of memories, including shared and pinned memory regions, without the requirement of page alignment for the registered memory buffer.

RDMA communications rely on three distinct types of queues: the Send Queue, Receive Queue, and Completion Queue. These queues, also known as Work Queues, serve as the foundation for managing and scheduling work in the RDMA framework. The Send Queue and Receive Queue are always created together as a pair, forming what is called a Queue-Pair

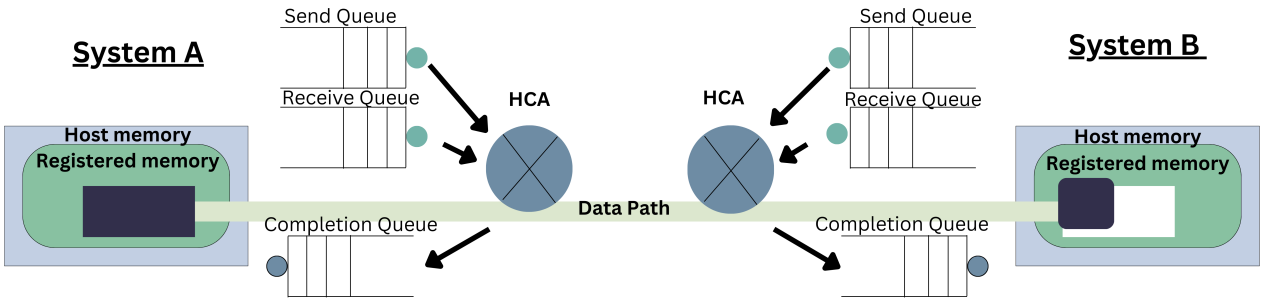
(QP). The primary function of these queues is to facilitate send/recv work instructions. In parallel, the Completion Queue (CQ) plays a vital role in signaling the completion of work requests within the Work Queue. When a work request is fulfilled, it is directly handled by the hardware within the Host Channel Adapter (HCA). The work requests, known as Work Queue Elements (WQE), convey important information to the HCA, such as the specific buffer to be sent or received. In the Send Queue, a WQE holds a pointer to the memory location that needs to be transmitted to the intended recipient. On the other hand, in the Receive Queue, a WQE contains a pointer to a buffer designated to hold incoming messages. Once a transaction is successfully executed, a Completion Queue Element (CQE) is generated and placed onto the Completion Queue. This element signifies the completion of a specific work task, allowing the application to track the progress and status of its RDMA operations effectively.

Figure 2.5: RDMA Operation involves Work Queues



When establishing a Queue Pair (QP) in RDMA, there are multiple transport modes avail-

Figure 2.6: RDMA Operation between two agents



able, each with its own characteristics -

1. Reliable Connection (RC): In this mode, a QP is associated with a single counterpart QP. Messages sent from the send queue of one QP are reliably received by the receive queue of the other QP. The packets are delivered in the correct sequence, ensuring reliable communication.

2. Unreliable Connection (UC): A QP in UC mode is also associated with a single counterpart QP. However, the connection is not reliable, making it a lossy transport. If a message encounters errors during transmission, the transport does not retry it. Any necessary error handling must be implemented at a higher level protocol.

3. Unreliable Datagram (UD): In UD mode, a QP can send and receive single-packet messages to and from any other UD QP. This mode does not guarantee ordering or delivery of packets, and the receiver may drop delivered packets. Additionally, UD

supports multicast messages, enabling one-to-many communication. This mode is similar to UDP (User Datagram Protocol) in traditional networking.

There are two types of verbs in RDMA: memory verbs and messaging verbs. The data transfer mechanisms encompass messaging verbs such as Sends and Recvs, and memory verbs such as RDMA Writes, RDMA Reads, and Atomics.

In the Send/Recv model, the source initiates a Send request, specifying the location of the data to be sent, while the destination posts a Receive request indicating where the incoming data will be written. A matching capability is used to associate a posted Receive with an incoming Send. This ”two-sided” interface ensures that each side possesses the necessary information for communication to occur.

In the RDMA model, which includes RDMA Write and RDMA Read, both the origin and destination buffers must be registered prior to any operations. Memory registration returns a handle that is used in RDMA operations, such as Read or Write (also known as Get or Put), to specify the origin or destination buffer. Unlike the Send/Recv model, only one side requires all the communication information, resulting in a ”one-sided” interface where one side is active and the other is passive.

2.7 RDMA over Converged Ethernet

RDMA over Converged Ethernet (RoCE) offers significant cost savings for data center providers by enabling RDMA without the need for new equipment or Ethernet infrastructure replacement[18]. To implement RoCE in a data center, network adapters or cards with RoCE support are installed, and RoCE drivers are available for major distributions, including Linux and Windows.

For network switches to support RoCE, it is necessary to select a switch with a network operating system that includes priority flow control.

There are two versions of RDMA over Converged Ethernet:

1. RoCE v1: RoCE v1 is an Ethernet link layer protocol that allows communication between two hosts within the same Ethernet broadcast domain (VLAN). It is a non-encrypted and non-routable RDMA protocol. RoCE v1 is suitable when non-encrypted RDMA is permitted in the environment.

2. RoCE v2: RoCE v2 overcomes the limitations of RoCE v1 by introducing packet encapsulation with IP and UDP headers. This enables the use of RoCE v2 across both Layer 2 and Layer 3 networks, allowing for routing and scalability across multiple subnets. RoCE v2 is also known as Routable RoCE (RRoCE). With RoCE v2, IP multicast is possible, expanding the capabilities of RoCE. Similar to RoCE v1, RoCE v2 is non-encrypted and requires permission for non-encrypted RDMA in the environment[6].

Additionally, there is an encrypted version of RoCE v2:

- Encrypted RoCE v2: This protocol adds encryption to RoCE v2 and provides data authentication, integrity, and confidentiality using IPsec protocols.

These different versions of RoCE provide flexibility and security options to accommodate various networking environments and requirements.

Soft-RoCE is a software implementation of the RoCE (RDMA over converged ethernet) protocol, which enables RDMA functionality over standard ethernet networks[3]. It consists of two main components: the ib_rxe kernel module and the librxe user space library. By utilizing Soft-RoCE, servers in data centers that are equipped with ethernet adapters but

lack dedicated RDMA hardware can establish connections with servers that utilize hardware-based RoCE.

The primary purpose of Soft-RoCE is to bridge the gap between systems with ethernet adapters and those with hardware-based RoCE capabilities. It allows a system with an ethernet adapter to communicate and inter-operate seamlessly with other devices that have either Soft-RoCE or hardware-based RoCE support. This capability enhances the connectivity options and facilitates efficient communication between diverse systems within a network environment. It effectively extends the benefits of RDMA technology to ethernet-enabled servers that would otherwise be limited to traditional ethernet communication.

Chapter 3

Related Work

A significant area for improvement in GPU Remoting software lies in reducing the communication overhead imposed by middleware. Several API Remoting software solutions, such as GiVM, vCUDA, GVM, and Pegasus, have been developed specifically for GPGPU applications but do not offer remote acceleration capabilities[11]. However, there are other software options available, including rCUDA[8], GVirtuS, Shadowfax, VOCL, and DS-CUDA, which do support remote acceleration.

Let's examine how communication is implemented in software that doesn't support remote acceleration.

vCUDA provides a CUDA wrapper library and virtual GPUs (vGPUs) in the guest OS, along with a vCUDA stub in the host OS. The wrapper library redirects API calls from the guest to the host, and each application has its own vGPU[17]. vCUDA utilizes VMRPC with VMCHANNEL for inter-VM communication, leveraging shared memory to reduce overhead. It also incorporates Lazy RPC to batch CUDA calls and optimize performance.

GViM focuses on efficient data sharing between the guest and host, using shared memory allocated by Xenstore[10]. GViM employs a one-copy mechanism that maps shared memory directly into the GPU application's address space, eliminating data copying and improving communication performance. Pegasus builds upon GViM, Pegasus and introduces the concept of an accelerator virtual CPU (aVCPU) to manage virtualized accelerators effectively.

Each aVCPU consists of a shared call buffer, a polling thread, and the CUDA runtime context. GPU requests are stored in the shared call buffer and fetched by polling threads, improving performance.

Finally, GVM utilizes a predictive performance model and includes user process APIs, the GPU Virtualization Manager (GVM), and virtual shared memory[13]. GVM exposes virtual GPU resources through APIs and operates in the host OS to handle requests from guest OSs. Communication between the guest and host OSs is facilitated by virtual shared memory, implemented as POSIX shared memory.

The trend is that almost all of them in a way use shared memory to improve communication performance between the guest and host operating systems.

3.1 Remote Acceleration Software

Shadowfax [14] builds upon its predecessor, Pegasus, by addressing the limitation of utilizing only local GPUs for applications requiring significant GPU computational power. To overcome this limitation, Shadowfax introduces GPGPU assemblies. This concept allows applications to span across multiple nodes, utilizing both local and remote GPUs. For remote execution, it implements a remote server thread that creates a simulated guest VM environment on the remote machine. The communication is conducted over a 1Gbps Ethernet fabric. Hence, it's worth noting that the latency for CudaMemCpy (both to and from the device) for 8 KiB of data is relatively high, approximately 100 ms over the network, compared to the actual latency of the call, which is slightly over 1 ms.

DS-CUDA is also a remote GPU virtualization platform that offers a solution for virtualizing GPUs[11]. It consists of a compiler that translates CUDA API calls into corresponding

wrapper functions and a server component that receives GPU calls and associated data through either an InfiniBand IBverb or RPC socket. One notable feature of DS-CUDA is its implementation of redundant calculations, which enhances reliability. This approach involves performing the same calculation on two different GPUs within the cluster to ensure the correctness of the computed results. DS-Cuda tries to group similar calls together to decrease communication time, but this limits the type of CUDA applications that support this approach.

Further on, VirtCL[22] and VOCL[21] introduce a GPU virtualization solution specifically designed for OpenCL applications. It enables virtual devices that support OpenCL by leveraging remote GPU-based acceleration. The VOCL framework consists of an OpenCL wrapper library on the client side and a VOCL proxy process on the server side. When a client application makes OpenCL calls, the wrapper library communicates with the proxy process via MPI (Message Passing Interface). The authors of VOCL assert that MPI offers a comprehensive communication interface and the ability to dynamically establish communication channels, making it a suitable choice compared to other transport methods. In terms of performance, the execution times of OpenCL application with two kernels using VOCL ranged from 1.3 to 1.7 seconds.

Among the various GPGPU Remoting software solutions, rCUDA[8] stands out as a recent and actively maintained system that receives significant attention in ongoing research. rCUDA is a remote GPU-based acceleration solution that facilitates the offloading of CUDA computations to GPUs located on remote hosts. It achieves this by implementing virtual CUDA-compatible devices without the need for a hypervisor layer. The system comprises a CUDA API wrapper library on the client side, responsible for intercepting and forwarding GPU calls from the client to the GPU server. On the server side, a server daemon receives and executes the remote GPU calls. Communication between the client and server

occurs via a TCP/IP socket. To address network performance limitations when multiple clients access the remote GPU cluster simultaneously, rCUDA incorporates a customized application-level communication protocol that is adapted to the underlying hardware. This protocol effectively mitigates bottlenecks and results in a 15% speedup for matrix operations with dimensions of 16k x 16k.

Moreover, the latest versions of rCUDA support RDMA over InfiniBand (IB), as well as TCP over Infiniband[19]. Notably, rCUDA execution over IB outperforms local CUDA counterparts by 10% to 85%, while surpassing CPU executions by up to 4.5 times, even when utilizing highly optimized libraries and 12-core computer configurations. The utilization of Mellanox adapters further enhances network bandwidth, contributing to improved performance in rCUDA deployments. Due to its closed-source nature, we were unable to utilize rCUDA for our evaluations and incorporate it into our research.

Chapter 4

Design & Implementation

The idea is to design a pluggable communicator, independent of the frontend and backend drivers which best suits the requirement for communicating CUDA APIs remotely. As mentioned in the previous section, TCP/IP stack exhibits poor performance and is more suited when using the software for academic purposes as compared to using it in a production data center. The design I propose specifically focuses on increasing the communication efficiency of GVirtuS.

4.1 Setup

In the initial phase of our project, we focused on setting up a functional environment to run GvirtuS using its original TCP/IP communicator. The setup involved configuring both the server and client machines.

The server machine we used is a CentOS Stream 8, x86 64-bit server with a Linux kernel v4.18. It is equipped with a Nvidia Tesla V100 GPU PCIe 16 GB, the client machine could be any Linux machine with an internet connection. In our case, we used an Ubuntu 20.04 LTS-based virtual machine or a CentOS 8 server without a GPU provisioned.

The physical links are 25 Gbit/s Ethernet with full duplex capacity. To support RDMA communication and accommodate large message sizes, we increased the MTU size to 4000

bytes.

In addition to installing the prerequisite software for GvirtuS, we installed RDMA packages on all devices with access to a physical RDMA-capable NIC. On the virtual machine, Soft-ROCE was installed to enable RDMA functionality. Alternatively, SR-IOV support could be utilized as an alternative to installing Soft-RoCE.

4.2 Design

RDMA is a highly tunable protocol, from the type of data transfer operations to the number and size of the hardware work queues, almost everything is customizable[12]. Some of the design decisions taken are based on hardware constraints, while some are recommended in heavily cited research papers. Along with these design decisions, to make the existing sequential API protocol more efficient, I designed a batching based approach that pipelines network transfer with on-device operations like GPU execution of the APIs or data copy from host memory to GPU memory.

4.2.1 Tuning RDMA & Connection Manager

The design choices made for RDMA systems, such as selecting specific RDMA operations and optimizing their usage, have a profound impact on overall performance[20]. Paying attention to low-level details is crucial in the design of RDMA systems to achieve optimal performance. Here are some decisions made in terms of which Verb APIs to use and other parameters while using RDMA and their reasoning :

1. Using channel based operations ie. RDMA Send/Recv for small messages which consists of Synchronous API over Memory based RDMA Read/Write operations[1]. This

can also be replaced with adding inline data in Work Queue Elements.

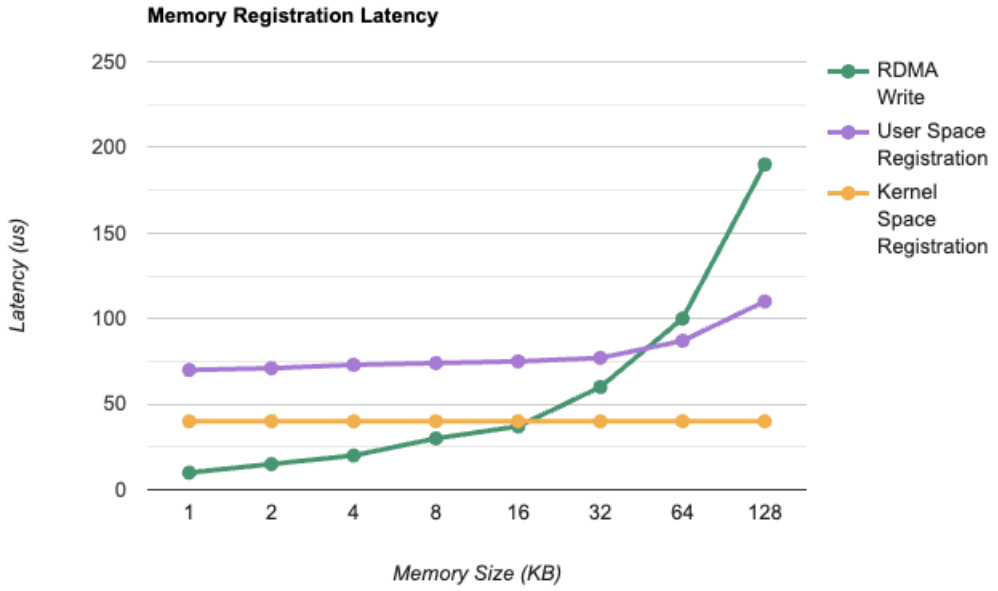
2. Message sizes are a tradeoff between memory consumption and communication efficiency. Small Sizes lead to more number of calls, which means more number of Work Requests and in turn leads to increase in CPU utilization. The maximum message size depends on the MTU size, however, larger message sizes require the application to pin and register substantially more memory. Memory registration latency is comparable to or higher (for large sizes of data) than the latency for a RDMA Write Operation. For 1 to 128KB, the latency is about 60 μs and increases linearly with size of the memory region. Moreover we perform these registrations in the userspace, given that the software runs in userspace, which also contributes to this high value.We undertake two approaches to deal with this, firstly we fix the size of the buffer to 1KB and increase the MTU size well over this value to avoid errors. Secondly and more importantly we remove the registration operations out of the data path and add them to the control path. Fig 4.1 shows user space memory registration latencies in the range of 75-100 μs which are higher than the RDMA Write operations for data values lower than 64KB.

3. Using Cache-line aligned buffers to decrease the number of CPU cycles and number of memory accesses. The cache-line size is 64 Bytes and hence we select 1024 as our communication buffer size.

4. Using Event-based Work Completion rather than polling based completion. This can help decrease CPU utilization and avoid unnecessary delays. In event based mechanism everytime an entry is added to the completion queue after the completion of a work request, a callback is executed.

5. The RC mode of RDMA provides a lossless and reliable mode of transport wherein the packet are delivered in order. Therefore the higher level protocol can focus on other

Figure 4.1: RDMA Memory Registration latencies



important aspect like queuing and memory.

6. The Size of work queues also contributes to protocol performance as seen in. As we will see in our protocol design in the next section, we limit the number of RDMA calls by transferring 1KB buffers of fixed size and hence the we keep the max size of our work queues on both ends as 4, equaling the number of our batching buffers.

7. We perform, pre-posting of RDMA receieve requests, rather than waiting for any event. Wherever possible we groups out post-send and post-receive operations to avoid delays, this is called the mailbox approach.

4.2.2 Protocol Design

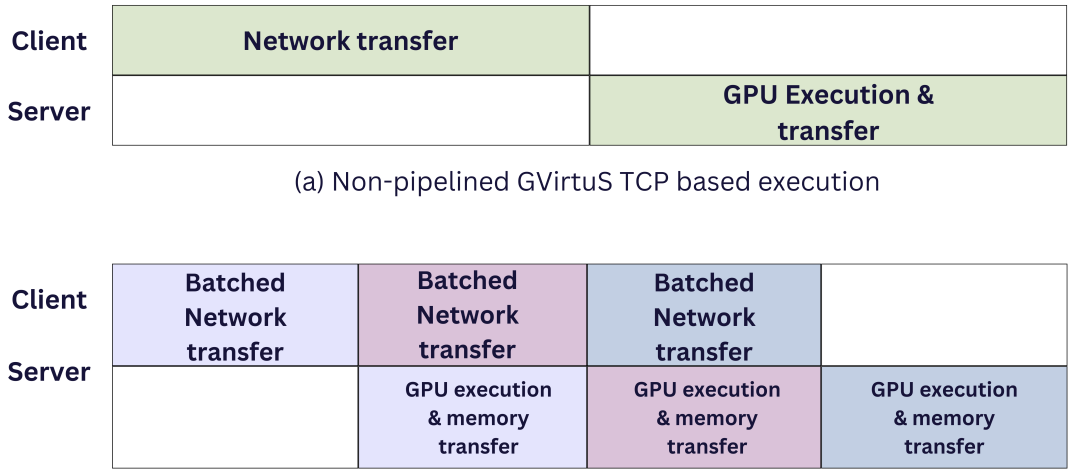
The Communicator is an application-level protocol used between frontends and the backend has been designed to be simple but effective, so that the interactions involve lesser processing and can efficiently leverage the RDMA network resources. Application data and control calls make use of this communication.

We pre-register the buffer memory on both the frontend and the backend side - making them stateful and cycle through the same set of buffers for every RDMA Send and Recv APIs in order to avoid large registration delays on every call. The buffers on the backend side are pinned memory so that they cannot be paged out allowing uninterrupted operation. Transfering data from the client directly to the pinned memory avoids one copy from pageable memory to pinned memory.

On the frontend, we have 4 registered buffers of size 1KB (1024 bytes) for sending data and 4 corresponding buffers of 128 bytes for their results. This is because most of the times the results will be return codes indicating success, failure or any other errors condition. Over on the backend side we have 4 registered buffers of size 1KB for receiving data form the frontend and 4 buffers of size 128 bytes each for sending the results back to the frontend.

We batch CUDA calls (intercepted by mock shared object files) and their arguments together to fill up the size of one registered buffer. When one buffer is completely full, we post a send request on the send-work queue. And while this request is being processed by the HCA and data is being read and calls being handled by the server from this buffer, we fill up the second buffer and follow this procedure. when we get a completion event for the first buffer, we clear it up for use by marking it as free. This way we pipeline the network transfer and the non-negligible processing involved on the frontend with the memory transfer and API call execution on the backend.

Figure 4.2: Pipelined Transfers of new design



We have a similar flow in the backend process, where our buffers receive data form the frontend, unpack them and CUDA APIs are executed in the received order. For most of the calls the return values are either success or error codes. For some calls like CudaMemCpy() when we need to copy data from GPU memory back to the device memory, the return values are much larger and may consist of multiple network transfers. We identify these calls in the mock libraries on the frontend that intercept them. While interfacing and identifying these calls, we mark calls that require large data transfers.

4.3 Implementation

GVirtuS is implemented in C++11 hence we used the same to implement the RDMA-based communicator.

First, we start with configuring RDMA Control Manager, that involve configuring RDMA

CM structures, this is the control path. Here we briefly describe the how we intially setup RDMA data and control path [5] -

1. Open a channel that will be used to report communication events with rdma_create_event_channel() which returns a pointer to the created event channel.

2. Next, we use the event channel to allocate a communication identifier which is used to track communication information.

3. We then get use the server IP and port number provided in the properties file and resolve the address using rdma_resolve_addr().

4. Next step is to allocate a protection domain for RDMA device context. This is followed by creating a completion event channel and a completion queue.

5. We then configure our application to be notified for completion events (as opposed to polling) by calling ibv_req_notify_cq() and providing the completion queue object created in the previous step.

6. Following this, we allocate our buffers and initialize them with required header information and then register this allocated memory with ibv_reg_mr().

7. This step is where perform most of our protocol tuning described in the previous section by configuring our Queue Pair attributes in a structure called ibv_qp_init_attr.

8. Finally after the setup and tuning steps we connect with the remote device using rdma_connect(). This is followed by posting of send and recv requests as described in detail below.

We have a multithreaded architecture on the frontend where for every instance of frontend (every program starts a new frontend instance) a new thread is spawned. If a previous

frontend instance already exists, it takes communicator information from it. Let's call this thread the "producer", it manages all the API calls and data information coming from the shared object files and is responsible for populating our batching buffers. We create a mutex for each registered buffer and the producer locks the corresponding mutex while populating data into it. In this critical section the producer thread copies data into the memory location and increments the pointer. Once the buffer is full, we use a counting semaphore to indicate that this buffer is available to be transmitted.

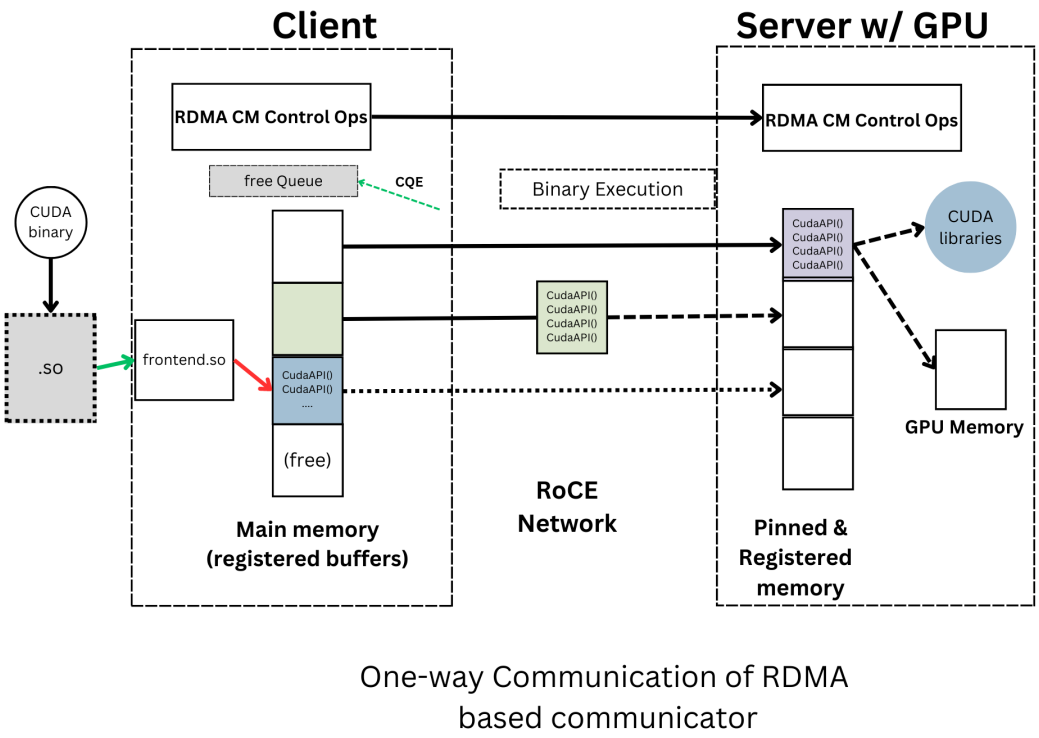
There is another thread, we can call this the network or consumer thread. It is responsible for invoking RDMA Send/Recv calls. It does so by generating the appropriate work request for a completely populated buffer that is ready to be transmitted. Following this posts the generated work request on the HCA queues. For every posted send request, we immediately pre-post a corresponding receive request for the result of the sent calls.

We register a callback function everytime we get a completion event for a work request. We keep the body of this function short and merely add the relevant buffer address to the free queue and mark it complete in the work request ID map.

All Standard C++ STL data structures used are thread safe. We use a queue to keep track of free buffers. We define buffers as free when they're either yet to be populated by API calls and data, or we have received a completion notification for them. We use an unordered map structure to keep track of work request IDs and their corresponding buffer address. Once a completion notification is received, the map entry is cleared.

Our buffers consist of the protocol header, followed by the payload. The 64 bit header consists of 3 fields. The work request ID (62 bits) that we generated while posting the work request, this is to keep track of our completions in our unordered map. The next two fields are required for a small subsets of CUDA calls like - CudaMemCpy, wherein the size of the

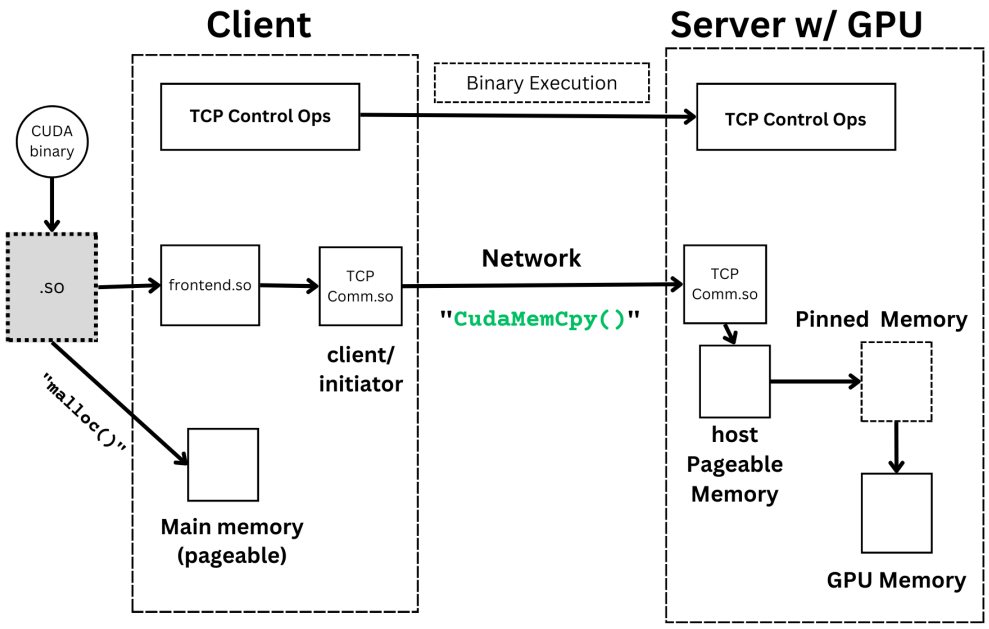
Figure 4.3: One - sided Communication of RDMA based Communicator



One-way Communication of RDMA based communicator

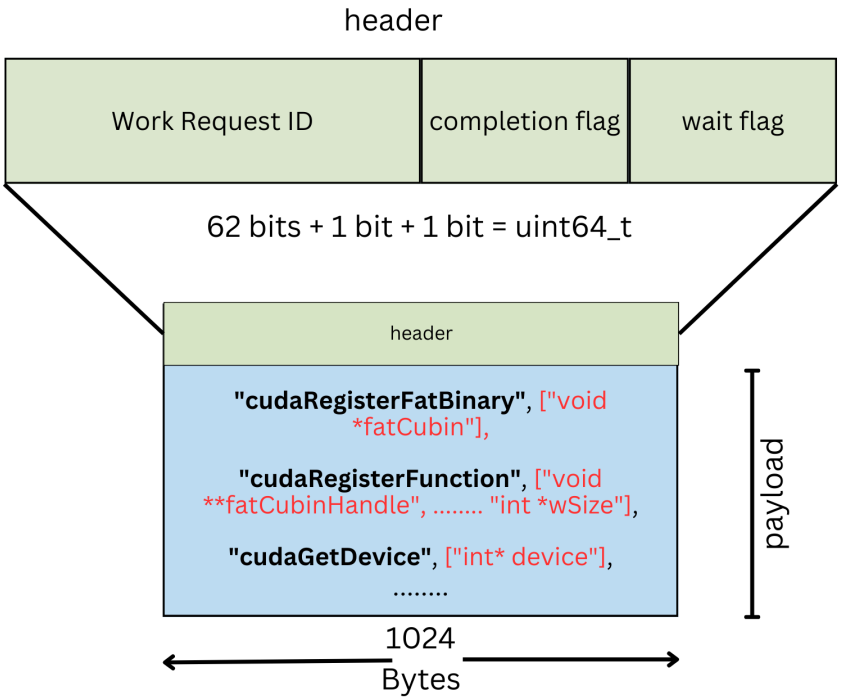
arguments is larger than the size of the registered buffers. In these cases, we set the the completion flag, indicating the buffer does not contain the full data to go ahead with the execution of the API. The wait flag is set to indicate that we need not wait for a completion event before sending the next packet. This is a slightly unreliable way of configuring the protocol and is used only we don't want to wait for the results of the previous APIs or we know the API has incomplete data. We follow a similar synchronization approach on the backend side wherein we have multiple threads managing the RDMA calls, the execution and the buffer management.

Figure 4.4: One - sided Communication of TCP based Communicator



One-way Communication of TCP based communicator

Figure 4.5: RDMA Batched Protocol Header and Payload



Chapter 5

Evaluation

To set some expectations before conducting evaluations on our communicator on our network and , we conducted tests to measure the base latency and throughput for both TCP and RDMA protocols.

Our network setup consists of a 25 Gbps Ethernet SFP28 network fabric with Mellanox Technologies MT27710 Family [ConnectX-4 Lx] 25 Gigabit Ethernet adapter cards that support RDMA over Converged Ethernet (RoCE v2).

For TCP evaluations, we utilized two benchmarking tools: sockperf and qperf. sockperf is a network benchmarking utility that tests the performance of high-performance and regular networking systems by measuring latency and throughput. It covers various socket API calls and options. qperf was used as a complementary tool to validate the sockperf results. The measured latency for a one-way path between the two machines was approximately 14 microseconds (µs), calculated as half of the round-trip time (RTT). sockperf provides latency measurements with sub-nanosecond resolution. The test configuration involved sending 338,000 messages with a message size of 1KB each, achieving maximum load. For the TCP throughput test, we used a message size of 1.4KB, resulting in a reported throughput of 227 Kbps and a message rate of approximately 20 messages per second.

Regarding RDMA evaluations, we employed the ibv_rc_pingpong test, which is suitable for Reliable Connection-based RDMA. With a message size of 8.192MB, we observed a latency

of 0.01 seconds, corresponding to a data transfer rate of 7275.31 Mbit/sec.

5.1 Execution Overhead

For our evaluations, we use a matrix multiplication benchmark. This benchmark was chosen as AI applications often involve a significant amount of matrix multiplications due to their reliance on mathematical operations performed on large datasets. Matrix multiplication is a fundamental operation in linear algebra and plays a crucial role in many AI algorithms, including deep learning. In deep learning, neural networks consist of interconnected layers of nodes (neurons) that perform computations on input data. These computations typically involve matrix operations, such as matrix multiplications and element-wise transformations, to process and transform the data as it propagates through the network.

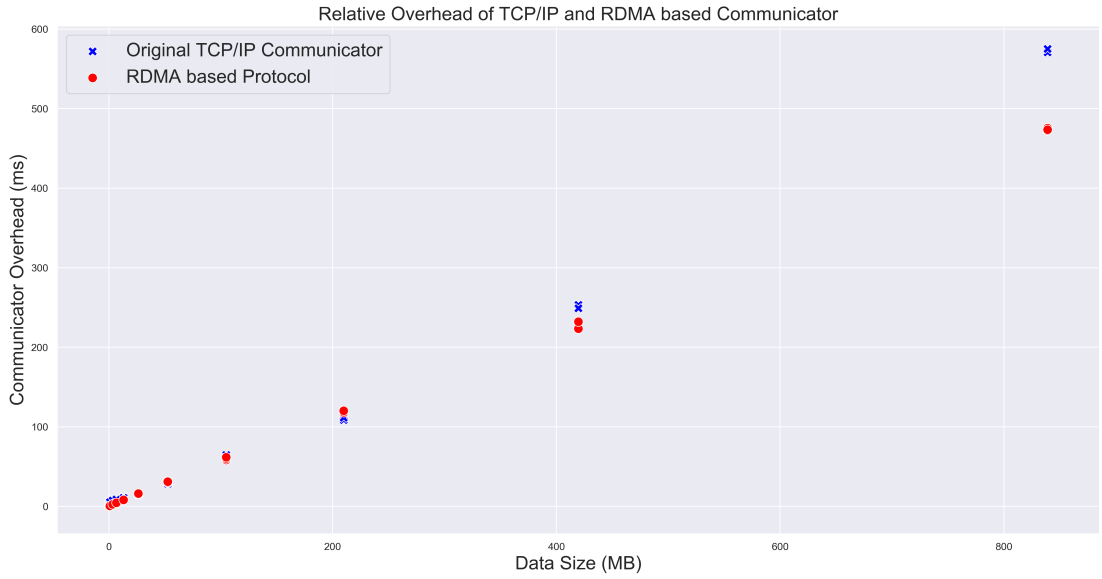
We conducted a series of experiments using various sizes of square matrices ranging from 160x160 to 10240x10240. This resulted in a total data transfer ranging from 262KB to 840MB (1GB) through the GVirtuS Communicator. This matrix multiplication example has 920 CUDA routines that are sent from the frontend to the backend which amounts to 50KB.

Our benchmarking procedure involved performing 300 iterations of matrix-matrix multiplication after an initial warm-up state. This approach was chosen to simulate the computational workload found in convolutional layers of the ResNet-50 model, as these layers often require multiple matrix multiplications. Along with this to measure the worst case latency and goodput numbers, we use Pageable memory copy calls in our code.

To measure the execution time, we employed the use of CudaEvents APIs. We placed CudaEvents markers around the execution kernel to indicate the start and end points of

the computation. Following the kernel execution, we used CudaEventSynchronize to ensure that all GPU operations were completed. Finally, we utilized the cudaEventElapsedTime function to calculate the elapsed time between the start and end events, providing us with the execution time for each iteration. Additionally, we employed a high-resolution steady clock on GVirtuS to validate the timing values.

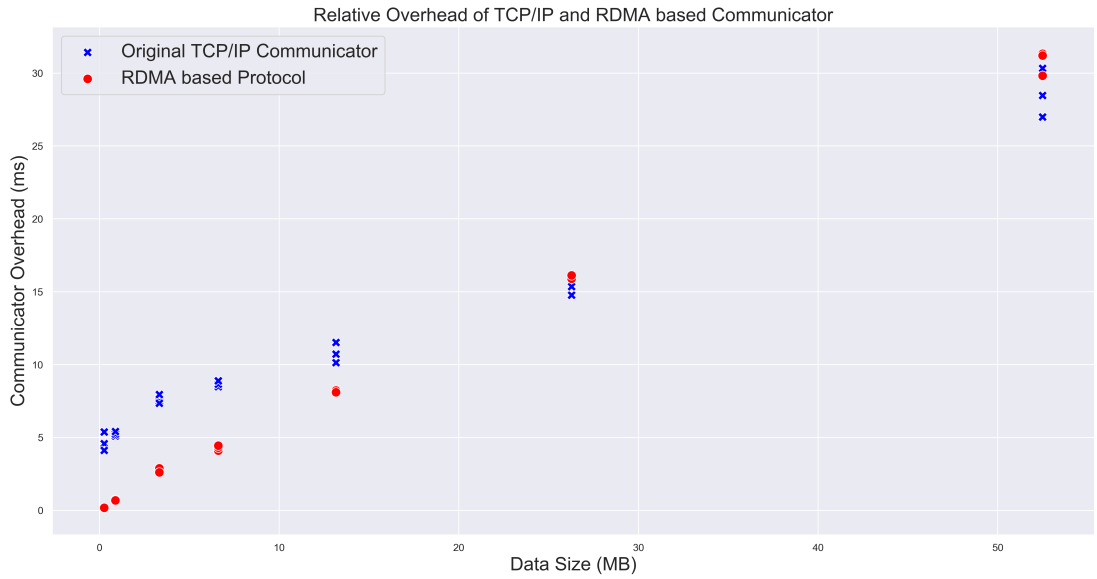
Figure 5.1: Relative Overhead of Communicator as compared to direct GPU execution - All data points



In our evaluation, we compared the relative time in milliseconds between the TCP and RDMA protocol-based communicators. This allowed us to assess the communication overhead associated with both communicators when compared to the execution time on the local device. Graph below depicts the trend in relative time in milliseconds (relative to on-device excution) of tcp and RDMA protocol based communicators. This could also be interpreted as the communication overhead of both the communicators. It is important to note the external factors contributing to these values, for example the RDMA Software Driver Soft-RoCE on

the VM adds to the timing values.

Figure 5.2: Relative Overhead of Communicator as compared to direct GPU execution - Lower data sizes

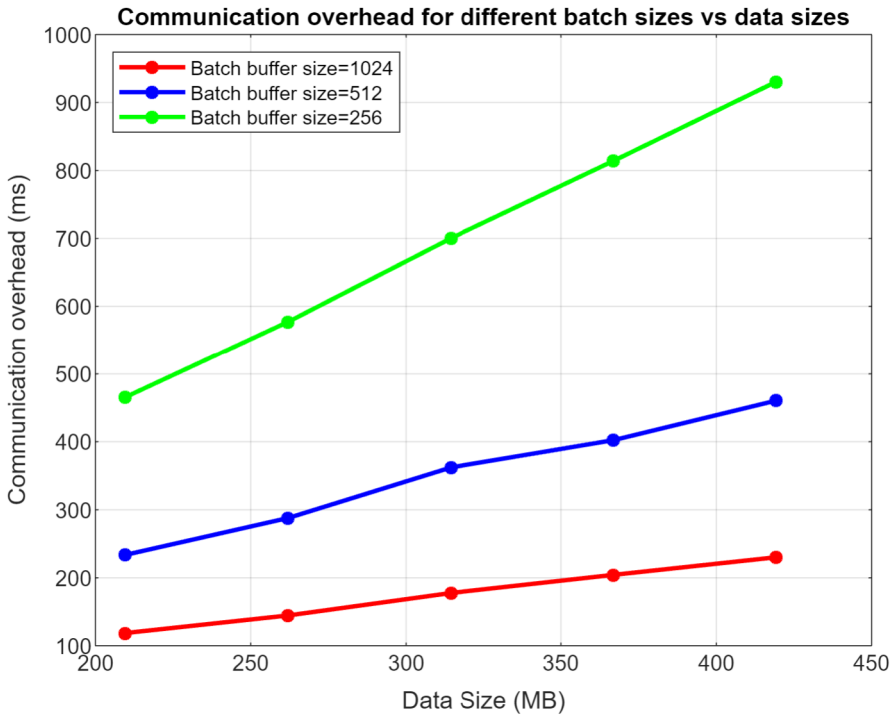


As we see in Fig 5.1 for data sizes over 300MB and over we see 10-17% reduction in time for our RDMA Protocol. But the trend is slightly different for the lower values of data. If we look at the graphs in Fig 5.2 we see that for data sizes 0.2MB upto 25 MB our RDMA based protocol outperforms TCP by an average of 50%. There is a inflection point at 30MB after which for data sizes upto 300MBs we see that the TCP version outperforms by 7%.This is most likely due to delays caused by synchronization required for the set of buffers used for batching on both frontend and backend.

5.1.1 Evaluation for varied Buffer Sizes

We vary the size of the buffer (used to batch the API calls and the data on both sides) and report relative timing values for the RDMA communicator observed for data sizes 200MB

Figure 5.3: Relative overhead for different buffer sizes for RDMA Communicator



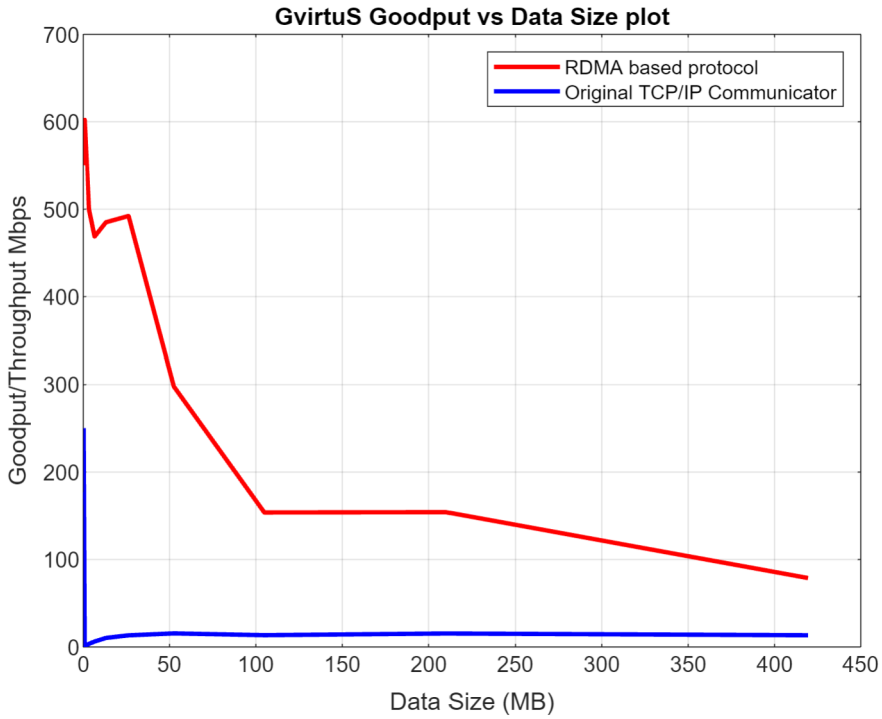
and above. As the size of the batching buffer reduces, the number of calls to be made increases.

5.2 Throughput

We calculate the goodput for both communicators by by measuring the total amount of data processed by the GVirtuS divided by the total time taken for execution over GvirtuS. For data sizes less than 1 MB, both communicators show high values of throughput. For higher sizes of data, the goodput when using the TCP/IP based communicator plateaus falling in the range of 5-25 Mbps. This is due to the sequential nature of communicator of GVituS along with the low througput of TCP protocol itself. If we observe the RDMA-based communicator values, we see that the throughput reduces with the increase in data sizes.

This can be attributed to various parameters.

Figure 5.4: Throughout/Goodput with both Communicators



It is important to note here that our transmission medium is the same for both protocols. Various factors specific to GVirtuS are likely contributors to the obtained throughput. Use of RDMA channel-based operations, size of buffers used for batching and synchronization-based delays.

Chapter 6

Conclusions

6.1 Limitations & Future Work

Our work attempts to reduce the communication latency in GPU Remoting. For this we employ a batching based RDMA protocol over a Ethernet medium. As this work is towards a Master's thesis, there were a lot of additional functionality that did not fit into the scope of this project.

1. Automating the scaling of buffers sizes and their count based on the executable sizes or size of the data used in the user program.

2. Integration with GPU Direct RDMA to avoid multiple copies from server CPU to GPU memory.

3. Adding support for for latest vesion of libraries like CuBLAS, CuFFT & CuDNN etc to support a larger set of user applications.

4. Adding support to compress data when sending over a network.

5. Testing the setup for a cluster of devices by adding support for SLURM. This would include directing client requests to a non-busy Backend Node.

6. Testing the RDMA protocol equipped GVirtuS with Infiniband hardware. RDMA

couples very well with Infiniband hardware giving an even higher bandwidth and 10 times lower latency as compared to TCP on Ethernet.

7. Providing container support on the client side. This would make the user experience a lot better but might lead to complexity in programming and also cause delays in execution as the container may not have direct support for RoCE.

6.2 Summary

API Remoting is a crucial approach to virtualize GPU resources for devices with low specifications like IoT devices, low-end client machines. In many cases only certain applications need to be accelerated and we don't need GPUs to be completely virtualized. API Remoting provided a use-per-need model of virtualization. Eventhough TCP/IP hardware is more commonly available and easy to deploy using the simple socket API calls, it is not suitable for high performance applications. This is due to many factors that contribute to it's reliability but also high per-packet delays involved in the kernel networking stack. RDMA or Remote direct memory access is a technology that can be used with existing ethernet infrastructure also known as RDMA over converged ethernet, which can used to read directly from a processes' virtual address space of the remote machine. It doesn't involved delays that are normally encountered in the TCP/IP networking. Along with adopting this protocol in an existing API Remoting software called GvirtuS, we also introduced a pipelined approach that uses batching of data and API calls grouping them based on their synchronous nature. We perform experiments to compare the performance of the newly designed protocol and the traditional TCP/IP based approach. It is evident from the evaluations that using batching based RDMA protocol that pipelines network transfer with GPU execution speeds up the communication and the overall execution time of running CUDA applications remotely.

Bibliography

[1] Cuda api synchronization behavior. https://docs.nvidia.com/cuda/cuda-driver-api/api-sync-behavior.html.

[2] Path of a packet in the linux kernel stack. https://www.cs.dartmouth.edu/~sergey/netreads/path-of-packet/Network_stack.pdf.

[3] Software based roce. https://www.roceinitiative.org/software-based-roce-a-new-way-to-experience-rdma/.

[4] Rdma over converged ethernet (roce). https://docs.nvidia.com/networking/pages/viewpage.action?pageId=39264632.

[5] Introduction to programming infiniband rdma. https://insujang.github.io/2020-02-09/introduction-to-programming-infiniband/.

[6] Roce version 2. https://docs.nvidia.com/networking/display/WINOFv55053000/RoCEv2.

[7] On the benefits of the remote gpu virtualization mechanism: The rcuda case. https://doi.org/10.1002/cpe.4072.

[8] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*, pages 224–231, 2010. doi: 10.1109/HPCS.2010.5547126.

[9] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In

European Conference on Parallel Processing, August 2010. URL https://api.semanticscholar.org/CorpusID:22345921.

[10] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing*, HPCVirt '09, page 17–24, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584652. doi: 10.1145/1519138.1519141. URL https://doi.org/10.1145/1519138.1519141.

[11] Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Comput. Surv.*, 50(3), jun 2017. ISSN 0360-0300. doi: 10.1145/3068281. URL https://doi.org/10.1145/3068281.

[12] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 437–450, USA, 2016. USENIX Association. ISBN 9781931971300.

[13] Teng Li, Vikram K. Narayana, Esam El-Araby, and Tarek El-Ghazawi. Gpu resource sharing and virtualization on high performance computing systems. In *2011 International Conference on Parallel Processing*, pages 733–742, 2011. doi: 10.1109/ICPP.2011.88.

[14] Alexander M. Merritt, Vishakha Gupta, Abhishek Verma, Ada Gavrilovska, and Karsten Schwan. Shadowfax: Scaling in heterogeneous cluster systems via gpgpu assemblies. In *Proceedings of the 5th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '11, page 3–10, New York, NY, USA, 2011. Associa-

tion for Computing Machinery. ISBN 9781450307017. doi: 10.1145/1996121.1996124. URL https://doi.org/10.1145/1996121.1996124.

[15] Raffaele Montella, Giulio Giunta, and Giuliano Laccetti. Virtualizing high-end gpgpus on arm clusters for the next generation of high performance cloud computing. In *Cluster Computing*, March 2014. URL https://link.springer.com/article/10.1007/s10586-013-0341-0.

[16] Raffaele Montella, Giulio Giunta, Valentina Pelliccia, Sokol Kosta, and Carmine Ferraro. Enabling android-based devices to high-end gpgpus. In *International Conference on Algorithms and Architectures for Parallel Processing*, November 2016. URL https://link.springer.com/chapter/10.1007/978-3-319-49583-5_9.

[17] Lin Shi, Hao Chen, and Jianhua Sun. vcuda: Gpu accelerated high performance computing in virtual machines. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, 2009. doi: 10.1109/IPDPS.2009.5161020.

[18] Alexander Shpiner, Eitan Zahavi, Omar Dahley, Aviv Barnea, Rotem Damsker, Gennady Yekelis, Michael Zus, Eitan Kuta, and Dean Baram. Roce rocks without pfc: Detailed evaluation. In *Proceedings of the Workshop on Kernel-Bypass Networks*, KBNets '17, page 25–30, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350532. doi: 10.1145/3098583.3098588. URL https://doi.org/10.1145/3098583.3098588.

[19] F. Silla, S. Iserte, Reaño Carlos, and J. Prades. On the benefits of the remote gpu virtualization mechanism: The rcuda case. *Concurrency and Computation: Practice and Experience*, 29(13), 2017. URL https://doi.org/10.1002/cpe.4072.

[20] Yong Wan, Dan Feng, Fang Wang, Liang Ming, and Yulai Xie. An in-depth analysis of tcp and rdma performance on modern server platform. In *2012 IEEE Seventh Interna-*

tional Conference on Networking, Architecture, and Storage, pages 164–171, 2012. doi: 10.1109/NAS.2012.25.

[21] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu chun Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. *2012 Innovative Parallel Computing (InPar)*, pages 1–12, 2012.

[22] Yi-Ping You, Hen-Jung Wu, Yeh-Ning Tsai, and Yen-Ting Chao. Virtcl: A framework for opencl device abstraction and management. *SIGPLAN Not.*, 50(8):161–172, jan 2015. ISSN 0362-1340. doi: 10.1145/2858788.2688505. URL https://doi.org/10.1145/2858788.2688505.