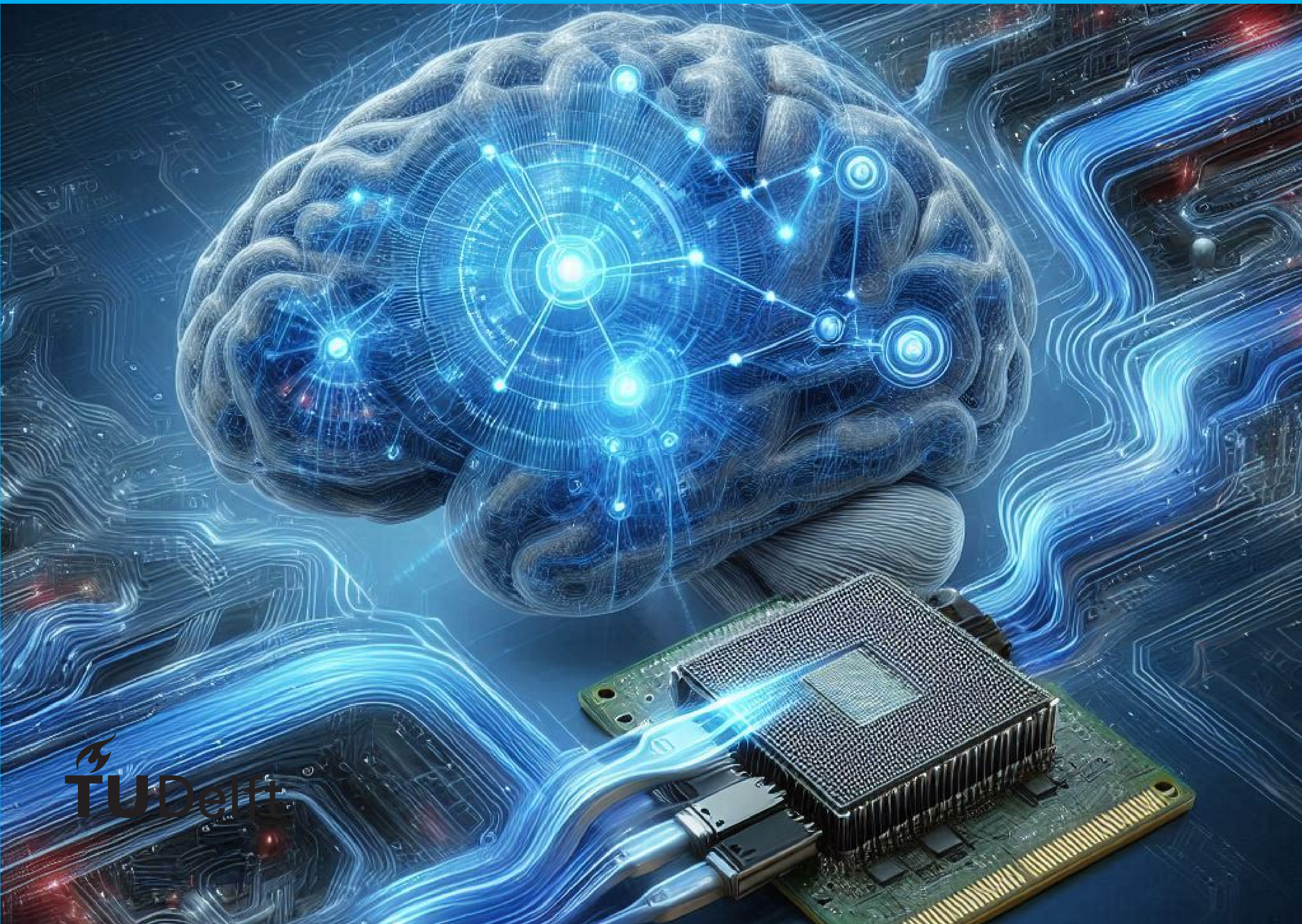
Extending the Computing Capabilities of Neural Interfaces

GPU Integration to the ONIX Electrophysiology System

A.J.R Bleeker

MSc. Thesis Report



Extending the Computing Capabilities of Neural Interfaces

GPU Integration to the ONIX Electrophysiology System

by

A.J.R Bleeker

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday January 27, 2025

Student number:	4713230	
Project duration:	March, 2023 – September 1, 2024	
Thesis committee:	dr. ir. C. Strydis,	TU Delft, supervisor
	dr. D.G. Muratore	TU Delft
	Prof. dr ir. G. Gaydadjiev,	TU Delft

Abstract

The exponential growth in the scale and complexity of neural recording systems demands increasingly efficient computational frameworks to manage the vast data generated by modern multi-electrode arrays. This thesis explores the feasibility of integrating Graphics Processing Unit (GPU) acceleration with the Open Ephys ONIX system to enhance its support for next-generation neural interfaces and computational algorithms. ONIX, a modular and open-source electrophysiology platform, currently relies on CPU-based computations that may struggle to meet the rising demands of closed-loop experiments.

This work investigates PCIe peer-to-peer (P2P) communication between ONIX and GPUs to bypass CPU involvement, aiming to reduce latency and increase throughput. A secondary data channel was implemented on the ONIX FPGA, accompanied by a custom kernel driver integrating GPUDirect with the existing RIFFA driver. Performance evaluations compared unpinned and pinned GPUDirect transfers to traditional CPU-mediated methods. Pinned transfers reduced transfer times by up to 30% for small data and 14% for larger data sizes, though significant variance in transfer time was observed. Application-level benchmarks revealed that GPU acceleration provides advantages for larger data sizes and higher computational loads but the effectiveness of GPUDirect is limited by ONIX's PCIe throughput.

The findings highlight that while GPUDirect offers theoretical benefits, its practical impact is constrained by hardware limitations, including the PCIe bandwidth of the ONIX FPGA. This thesis outlines necessary hardware improvements, such as higher-speed SerDes modules and advanced FPGA cards, to fully realize the potential of GPU acceleration in electrophysiology systems. A decision tree framework is proposed to guide researchers in determining scenarios where GPU and GPUDirect integration is beneficial. While this study demonstrates the feasibility of GPU integration with ONIX, GPUDirect integration is currently not a beneficial technology for the ONIX ecosystem.

Acknowledgements

I would like to sincerely thank my thesis advisor, Christos Strydis, for his guidance and constructive feedback throughout this project. His advice and enthusiasm have been crucial in helping me navigate to the end of this work.

I am also thankful to the members of the Neurocomputing Lab for their support and insights during my time at TU Delft. The collaborative environment and willingness to help made a significant difference.

To my friends I made at TU Delft and before, thank you for all the good times, fun nights and trips we made during my studies.

Lastly, I would like to express my gratitude to my family and my girlfriend. Their encouragement, patience, and understanding have been a constant source of motivation, and I am deeply appreciative of their support.

A.J.R Bleeker
Delft, January 2025

List of Acronyms

BMI	Brain-Machine Interface	1
CPU	Central Processing Unit	1
GPU	Graphics Processing Unit	1
PCIe	Peripheral Component Interconnect Express	
API	Application Programming Interface	7
FPGA	Field Programmable Gate Array	5
MMU	Memory Management Unit	12
IOMMU	Input Output Memory Management Unit	12
DMA	Direct Memory Access	
MSI	Message Signaled Interrupt	13
GPGPU	General-Purpose computing on Graphics Processing Units	2
AI	Arithmetic Intensity	41
P2P	peer-to-peer	2
PCA	Principle Component Analysis	1
ONI	Open Ephys Network Interface	5
IMU	Inertial Measurement Unit	6
SD	Standard Deviation	30
FLOPs	Floating Point Operations	41
GFLOPS	Giga Floating Point Operations	41
SMs	streaming multiprocessors	43
BAR	Base Address Register	25

Contents

1 Introduction **1**
1.1 Problem statement 1
1.2 Motivation 1
1.3 Thesis Goal 2
1.4 Thesis Outline 3

2 Background and Related Work **4**
2.1 ONIX data acquisition system for ephys experiments 4
2.2 System Overview 5
2.3 Application and Devices 5
2.4 FPGA logic design 6
2.5 Liboni and API 7
2.5.1 Initialization 8
2.5.2 Device interaction 8
2.5.3 Reading and Writing Data 8
2.5.4 Driver development for Liboni 10
2.6 PCIe and Kernel driver 10
2.6.1 PCIe Bus Organization 10
2.6.2 Base Address Registers (BAR) Space in PCIe 10
2.7 Linux Kernel & Memory Spaces 12
2.7.1 Scatter-Gather and DMA 13
2.8 RIFFA 13
2.9 Graphics processing unit 15
2.9.1 CUDA 16
2.10 GPUDirect RDMA 16
2.11 Related work 17
2.11.1 Ephys Systems 17
2.11.2 Ephys Software 18
2.11.3 Direct FPGA-GPU Communication 18

3 Design and Implementation **21**
3.1 Base Case 21
3.2 Reducing Transfer Steps 22
3.3 Direct PCIe transfer 23
3.4 ONIX FPGA-Core Modification 24
3.5 Linux RIFFA-Kernel Driver Modification 25
3.5.1 Wrapper Kernel Module 26
3.6 Liboni Library Update 27
3.7 Conclusion 27

4 Performance Evaluation **29**
4.1 Experimental Setup 29
4.2 FPGA testing logic 30
4.3 Performance of Original System 31
4.3.1 Results 31
4.3.2 Conclusion 32
4.4 Performance of Updated system without GPUDirect 32
4.4.1 Results 33
4.4.2 Conclusion 33

4.5 Performance of GPUDirect without Prepinning 34
4.5.1 Results 34
4.5.2 Conclusion 34
4.6 Performance of GPUDirect with Prepinning 35
4.6.1 Results 35
4.6.2 Conclusion 36
4.7 Profiling kernel driver 38
4.7.1 Results 38
4.7.2 Conclusion 38
4.8 Conclusion of performance evaluation 39

5 Application Scenarios **40**
5.1 Implementation of Application Scenarios 40
5.2 Results 43
5.2.1 Small Data Transfer, Small Computation 43
5.2.2 Small Data transfer, Large Computation 43
5.2.3 Large Data Transfer, Small Computation 44
5.2.4 Large Data Transfer, Large Computation 45
5.3 Conclusion 45

6 Discussion and Future Directions **46**
6.1 Cases for Utilizing GPUDirect 46
6.2 Potential Upgrades to ONIX System Components 46
6.3 Comparison of GPUDirect Solution to Previous Work 47
6.4 Conclusion 49

7 Conclusion **51**
7.1 Summary 51
7.2 Main Contributions 52
7.3 Future Work 52

1 Introduction

1.1. Problem statement

In recent years, there has been a significant increase in the number of recorded neurons, mirroring the trend of the number of transistors in integrated circuits following Moore's law, as depicted in figure 1.1 [34]. The figure shows that the number of neurons recorded in experiments doubles approximately every 7 years. The leading technology in neural recording has been mainly electrode arrays [11]. A multi-electrode array is a grid of tightly spaced microscopic electrodes. With these electrodes, it is possible to measure the neural activity of the brain at many sites. Using measurements, individual neuron responses and the way an ensemble of neurons interact with each other can be measured. These measurements shed light on the neural networks involved in various tasks, such as sensory responses, motor activities, and cognitive processes, allowing the development of an efficient Brain-Machine Interface (BMI).

However, the algorithms used to find these networks can be quite demanding. Examples of this are spike sorting [5] to find the activation of individual cells or dimensionality reduction algorithms like Principle Component Analysis (PCA) to find latent variables in the data [21]. As the electrode count increases, the computational demand for these algorithms also increases. In the case of BMI, it is especially important that these algorithms run fast enough. After the measured activity, a response must be quickly sent back to ensure correct circuit behavior. Thus, it is pivotal that computational requirements can scale with the increasing number of neurons measured.

Traditionally, the Central Processing Unit (CPU) has served as the primary computing platform in closed-loop experiments due to its versatility. However, with escalating computational demands, particularly for memory-intensive tasks such as machine learning, alternative hardware targets must be considered. The Graphics Processing Unit (GPU) present a compelling solution, offering a multitude of parallel cores and ease of interface and programming [7]. Due to the fact that GPUs have a much higher core count than CPUs, both the memory bandwidth and computational throughput of a GPU tends to be higher than that of a CPU. However, algorithms should be optimized such that they can fully utilize the GPU hardware.

1.2. Motivation

Electrophysiology, often abbreviated as ephys, is the study of electrical properties of biological cells and tissues. Many ephys systems have been developed to facilitate neural recordings in laboratory settings. ONIX is one such system [24], developed by the Open Ephys project, and has gained popularity due to its open-source nature and streamlined support for a variety of neural interfaces, such as the Neuropixels probes and other electrode arrays [9]. ONIX is a combination of hardware and software designed to seamlessly integrate with diverse neurological experiment setups. In particular, it enables both data acquisition and stimulation, making it suitable for closed-loop brain experiments.

ONIX is built with a modular architecture that can handle significantly higher data throughput as neural interface technology advances. It contains a reconfigurable FPGA, which can be programmed to manage complex data processing tasks efficiently. The FPGA enables ONIX to handle the increasing data volumes generated by neural recording devices as electrode counts and sampling rates rise.

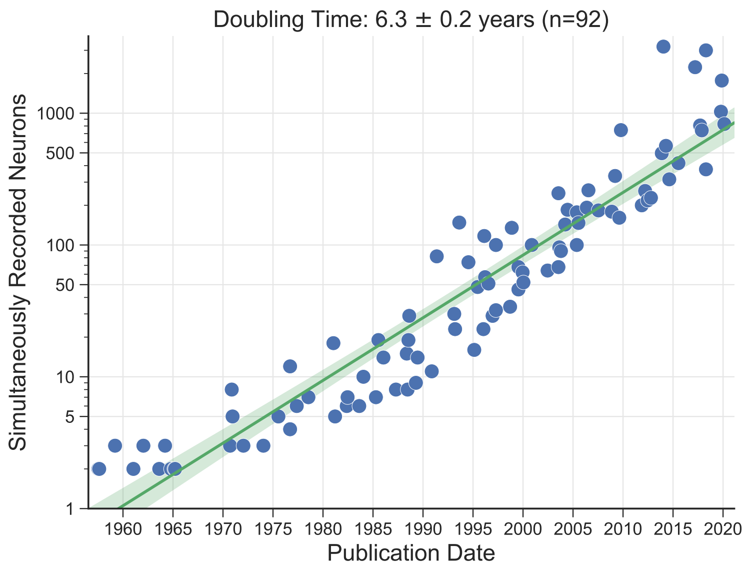


Figure 1.1: Trend of recorded neurons in experiments from published papers [34].

Additionally, ONIX connects to the PCIe bus, allowing for high-speed data transfer between the system and other components. The current configuration of ONIX limits computation to CPUs. As mentioned earlier, introducing GPUs into the computational framework presents an opportunity to run more complex algorithms, thus offering deeper insights into neural mechanisms while keeping pace with these increasing data demands.

Using GPU resources has become relatively straightforward due to the availability of hardware access through General-Purpose computing on Graphics Processing Units (GPGPU) libraries such as CUDA and OpenCL. However, a notable drawback of these libraries is that data must first be collected by the CPU before being sent to the GPU. This intermediate step could potentially be eliminated. Both the ONIX system and the GPU are connected to the PC via a PCIe bus, which allows for the possibility of direct communication between them. In applications such as closed-loop BMI systems, minimizing latency is crucial for proper system operation. Consequently, PCIe peer-to-peer (P2P) communication could be a valuable enhancement for the ONIX system, particularly as the demand for higher throughput and lower latencies is expected to grow in the future.

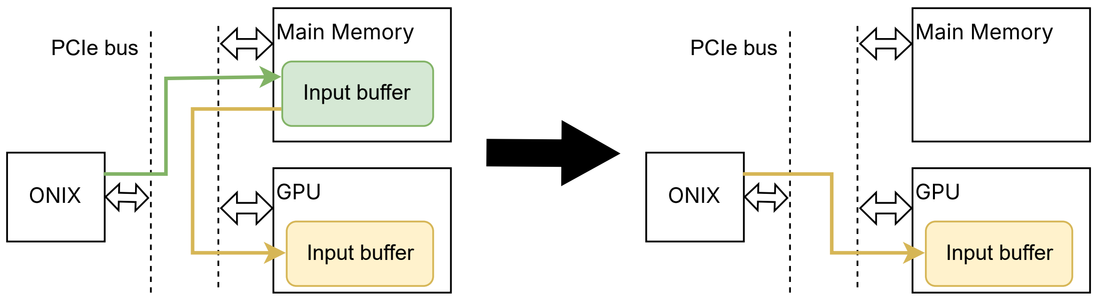


Figure 1.2: Sending data to the GPU doing computation and sending it back. First without PCIe P2P and after with PCIe P2P.

1.3. Thesis Goal

The primary goal of this thesis is to extend the existing ONIX neural recording system to make it suitable for the next generation of neural interfaces and computational algorithms. This involves ensuring that the system can sustain higher data rates and support more complex processing demands, particularly in the context of both open-loop and closed-loop experiments. The focus will be on integrating advanced hardware and software solutions that can handle the growing number of channels, higher data acquisition speeds, and increasingly powerful algorithms used in neural interfaces.

How can GPU computation be effectively integrated with the ONIX electrophysiology system to enhance support for next-generation neural interfaces and computational algorithms?

To achieve this goal, the following specific objectives are defined:

- Analyze the technology stack that underpins the ONIX system and related technologies.
- Determine the necessary modifications to enhance data throughput and reduce latency in the ONIX system, enabling support for high-speed neural interfaces.
- Develop and implement hardware and software changes in the ONIX platform to integrate GPU acceleration effectively.
- Evaluate system performance with and without GPU acceleration, to determine if the proposed enhancements deliver improvements.
- Investigate future hardware changes to the ONIX system to more effectively handle higher data loads.

1.4. Thesis Outline

In Chapter 2, background information on the ONIX system and the technologies required for GPU integration is presented. A comprehensive understanding of the ONIX system is essential to determine the necessary modifications. At the end of the chapter, a review of related work is performed, exploring other electrophysiology systems and software to understand their technology stacks and assess how GPU acceleration has been implemented in comparable systems.

With the foundational background and related works in place, Chapter 3 will go over all the necessary hardware and software modifications that are implemented in the ONIX system to try and establish GPU integration, addressing the second research goal. In Chapter 4 the performance of the system is evaluated, and tests will be conducted to assess whether the modifications improve latency and performance. Subsequently, in Chapter 5 realistic usage scenarios of the modified ONIX system are simulated and the aggregate performance is evaluated. Chapter 6 discusses how the implemented technology might be used in the future. Furthermore, an investigation is done on how the hardware of the ONIX system could be improved in the future to handle increasingly large neural interfaces. In Chapter 7, a thesis outline is provided, which summarizes the main contributions of the work and discusses potential future research directions and emerging opportunities based on the findings of this study.

2 Background and Related Work

In this chapter, the necessary background information on the ONIX system is provided, which is essential for understanding the design modifications and performance evaluations presented in later chapters. The system's architecture is explained, including its libraries and hardware components, and detail how they adhere to the ONI standard. This foundational knowledge is critical to ensuring that the integration of a GPU does not interfere with ONIX's modular and flexible framework. Additionally, this chapter reviews the methods of data transfer to a GPU and kernel execution to identify the most effective approaches for balancing latency, throughput, and usability. Finally, a survey is done of related literature and prior work that explores similar technological integrations and improvements in electrophysiology systems. This review provides context and helps frame the contributions of this thesis in the broader field of neural data acquisition and processing.

2.1. ONIX data acquisition system for ephys experiments

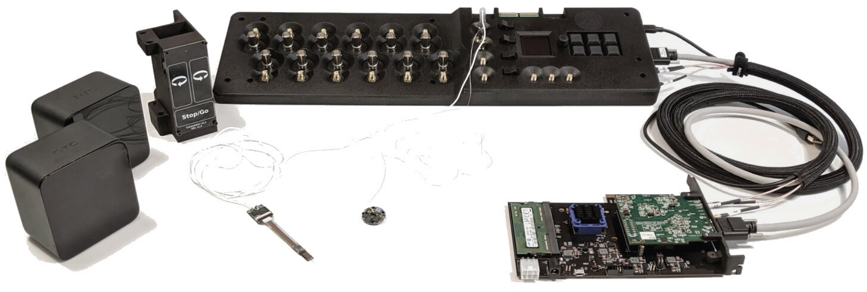


Figure 2.1: Overview of main hardware components in a typical ONIX system [10]

The ONIX system is an open-source platform designed to facilitate data acquisition for neuroscience experiments, particularly those involving electrophysiology (ephys). Its modular design allows it to support a wide variety of experimental devices, ranging from extracellular probes to cameras and tracking systems, all of which can be connected to a single acquisition platform. By standardizing data organization while remaining flexible about the type of data being acquired, ONIX aims to create a unified system for neuroscience research. This standardization allows researchers to use the same infrastructure for different types of neural recordings, reducing the need for multiple acquisition systems and promoting interoperability across experiments.

The main component of ONIX is the PCIe host card, which serves as the central hub for data acquisition and processing. As shown in Figure 2.1, this host card is composed of a Xilinx Kintex-7 FPGA-based Nereid PCI Express development board, along with a custom PCB that interfaces with it via an FMC header. The custom PCB is designed to connect various experimental devices using coaxial cables, while the FPGA board provides the high-speed processing necessary for handling large data volumes in real time.

The FPGA in the ONIX system plays a critical role by managing the data flow between connected devices and the central PC, using the PCIe interface for high-throughput data transfer. The Nereid Kintex-7 board, with its x4 PCIe interface and 4GB of DDR3 SDRAM, is responsible for transmitting the data to the host computer.

ONIX is part of the broader open-ephys initiative, which promotes the development of affordable, open-source tools for electrophysiology experiments. Open-ephys has made the ONIX system popular due to its adaptability and compatibility with various neural interfaces and sensors. This modular approach, combined with its open-source nature, has facilitated widespread use in the research community, allowing for continual improvements and customization. In this thesis, we explore how GPU computation can be integrated into this framework to support the growing data demands of next-generation neural interfaces, without disrupting the core principles of the ONIX system.

2.2. System Overview

As shown in Figure 2.2, the system consists of three main layers: the application and devices, the Liboni library and the ONIX Field Programmable Gate Array (FPGA) Core and finally the PCIe communication library RIFFA [29]. The application allows the user to read and write data and modify the configuration parameters of the devices and the CPU. The application interacts with the Liboni API, which is a high-level interface that abstracts away the details of the underlying communication protocol. The Liboni API is implemented using the Liboni library, which is a C implementation of the Open Ephys Network Interface (ONI) standard. The ONI standard defines a common protocol for data acquisition and communication between devices and software in neuroscience experiments. The Liboni library communicates with the ONIX FPGA Core, which is a hardware module that multiplexes the data streams from different devices and sends them to the CPU. The communication between the Liboni library and the ONIX FPGA Core is based on RIFFA, an open-source kernel driver and FPGA core that enables easy communication over PCIe using simple read and write calls [29]. However, the ONI standard is designed to be independent of the communication protocol, enabling the use of other protocols as well.

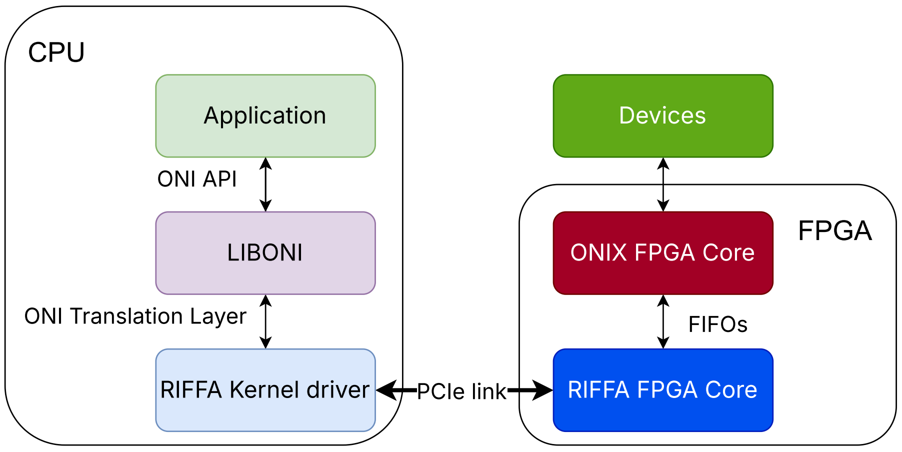


Figure 2.2: System overview of ONIX

2.3. Application and Devices

The parts that the user interacts with at both ends of the system are the application and devices. ONI tries to abstract everything that happens between them away. The platform is agnostic to the application attached to it, as long as the application uses the ONI API to communicate with the library. Currently, the best-supported application is Bonsai, an open-source framework that enables the visual programming of control loops for various asynchronous data streams [20]. The library also provides C/C++ and C# APIs for custom user applications.

The devices compatible with the platform are primarily headstages, which are small circuit boards that amplify and digitize neural signals from electrodes. The headstages can interface with the PC card using a serial communication link over a coaxial cable. For this communication, the DS90UB933 [12]/DS90UB934 [13] pair is used. Each headstage that the ONIX supports should have a DS90UB933 serializer on the headstage side. On the PC card side, two DS90UB934 deserializers are present. The SerDes pair supports a maximum data rate of 150 MB/s; hence with two deserializers, the system can handle a maximum input data rate of 300MB/s.

The supported device with the highest data rate is currently the Neuropixel V2 [33], a high-density sensor with up to 5120 electrodes. It can record 384 12-bit channels simultaneously at a sampling frequency of 30 kHz, resulting in a data rate of 138 Mbps.

The card also supports the Miniscope, a head-mounted CMOS imaging sensor used in calcium imaging [6]. Other types of devices the platform supports include an Inertial Measurement Unit (IMU) and a position tracker, which can provide additional data about the motion and orientation of the subject. The devices have multiple inputs for analog and digital I/O, which can be utilized to transmit and receive signals from other devices or stimuli.
To incorporate new devices, these need to be integrated into the FPGA VHDL code, the hardware description language that specifies the logic of the platform. The code for this is proprietary but can be obtained if the end goal of the user is non-profit. The platform can accommodate up to 256 devices, each with a unique ID and a data rate. Table 2.1 lists some of the devices that can be connected and their respective data rates.

Device	**Peak Data Rate (Mbps)**
Neuropixel V1	115 [22]
Neuropixel V2	138 [33]
BNO055 (IMU)	2.8 [31]
UCLA Miniscope V4	90 [6]

Table 2.1: Example of supported devices in the ONIX system and their peak data rates.

2.4. FPGA logic design

The interface between the devices and the PC is handled by the FPGA, which stands for Field Programmable Gate Array. This is a chip where the logic can be reprogrammed so that any logic function fitting within the hardware can be realized. The logic can be described using a hardware description language such as VHDL or Verilog. To compile the code into an implementation which can be programmed on the device, a synthesizer is used. Creating custom logic for a specific function allows it to be executed with very low latency and high throughput, making the FPGA an ideal platform for the data acquisition context of ONIX. Figure 2.3 presents a simplified diagram of the logic structure programmed on the FPGA.

Within the logic, there is an implementation for each device that can be connected to the ONIX card. These devices handle reading and writing to device registers for configuration, as well as managing incoming and outgoing data. This is the interface the user interacts with to indirectly communicate with the devices connected to the system. To send and receive data, each device has a custom frame with a fixed header, as shown in Figure 2.4.

The next part of the core is the data input, which acts as a multiplexer, passing data from the devices into a single data stream in a round-robin fashion. Additionally, it adds headers to the data from each device, creating single frames that the application can read.

The deep FIFO functions as a backup buffer for the read stream. When the FIFO in the read stream block is full, incoming data is stored in the DDR module of the FPGA, which has a capacity of 4GB. Otherwise, it simply passes the data to the write stream.

The data output performs the opposite function of the data input by demultiplexing the single data stream. It reads the headers of the data and sends it to the correct device.

The Config controller is responsible for configuring both the individual devices and global settings.

It can also be used to read current configuration values. The signal controller has two main responsibilities: after the FPGA is reset, it sends all the devices present in the system to the library, and when the application sends a configuration request to the FPGA, the signal controller sends an acknowledgment to indicate whether the request was successful.

The write stream serves as the interface between the logic and the RIFFA core. The RIFFA core will be explained in more detail in 2.8. It waits until the user wants to receive data, at which point it writes a block of data to the user. The size of this block can be adjusted by writing the desired value to the write stream. All other streams also serve as interfaces between the logic and RIFFA, containing a FIFO buffer.

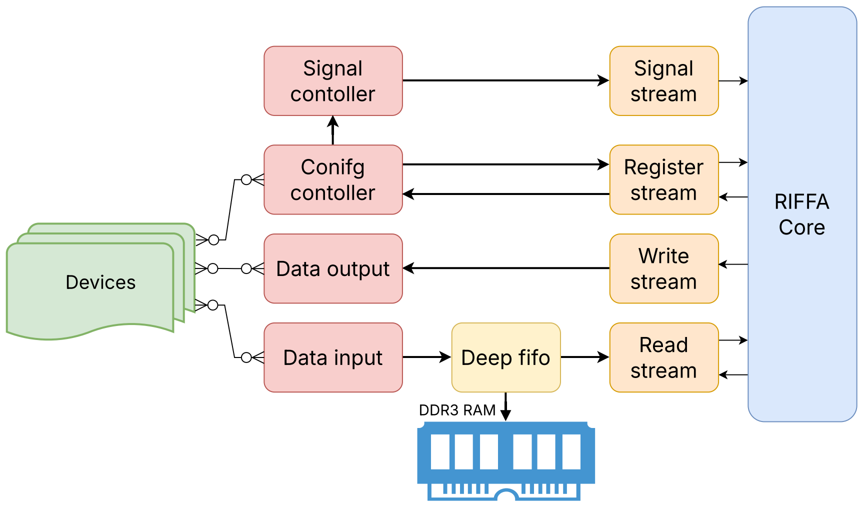


Figure 2.3: Overview of the logic on ONIX FPGA

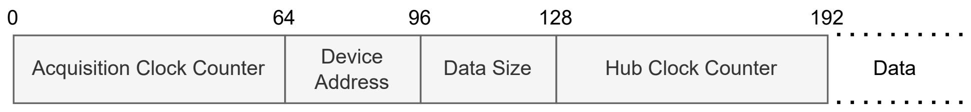


Figure 2.4: Frame Header Structure

2.5. Liboni and API

The Liboni serves as an interface layer between the application and the hardware. It provides a simple and consistent Application Programming Interface (API) for creating and running ONI-compatible experiments. Additionally, it offers a standardized interface for writing hardware drivers, allowing flexibility in the underlying communication protocol between the PC and the ONIX card.

The API is composed of several key components:

- `oni.h`: Defines the main functions and data structures of the library.
- `onidriver.h`: Specifies the interface for the hardware communication layer.
- `onix.h` (optional): Includes ONIX-specific features.
- `onitest.h` (optional): Provides tools for testing and debugging purposes.

This structure ensures that developers can efficiently interface with ONI hardware while maintaining the flexibility to adapt to various communication protocols and specific use cases.

2.5.1. Initialization

During the initialization phase, Liboni sets up the necessary components to enable communication between the application and the hardware. The hardware state is represented by a structure called `oni_ctx`. Users can obtain this structure by invoking the `oni_create_ctx` function. This function dynamically loads the desired driver for hardware communication, such as RIFFA [29], Xillybus [3], or FTDI USB3.0 and returns an `oni_ctx` structure. Subsequently, the `oni_ctx_init` function can be called using this structure, which retrieves the device table from the hardware and configures the read and write buffers on the PC.

The device table is a data structure that specifies the devices connected to the ONIX card. Interaction with `oni_ctx` is facilitated by the `oni_get_opt` and `oni_set_opt` functions. These functions allow various actions, such as reading the device table or initiating a system reset, to be performed by the user application. An overview of the initialization process is provided in Figure 2.5.

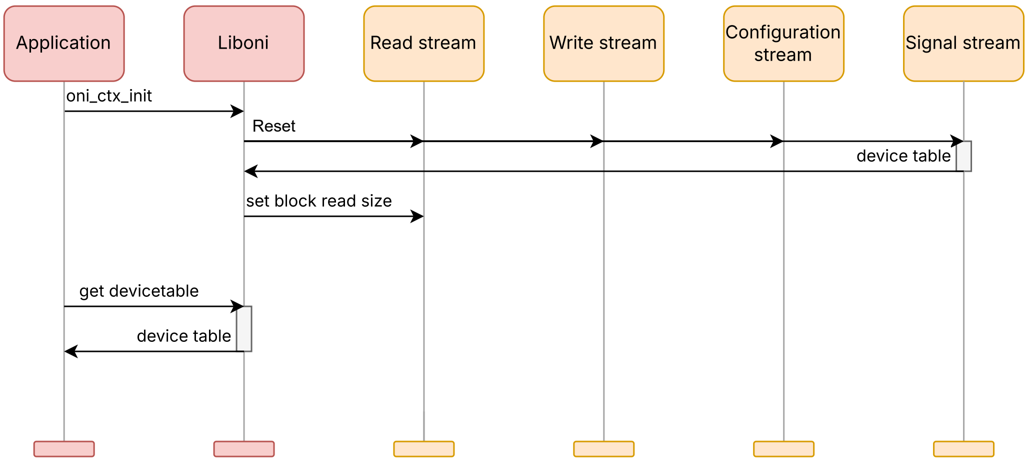


Figure 2.5: Overview of initialization in Liboni.

2.5.2. Device interaction

After proper initialization, the user can interact with the devices through the use of registers. The API has two functions to interact with these registers: the `oni_write_reg` and `oni_read_reg`. A call to these functions will cause a write to the configuration stream. In case of a sucessful write, the signal stream will return an acknowledgment. For reads, the signal steam will first acknowledge that the read is allowed, then the library will return the value from the configuration stream and then return it to the user.

2.5.3. Reading and Writing Data

Once everything is set up correctly, the device can be signaled to start data acquisition by calling `oni_set_opt` with `ONI_OPT_RUNNING`. This action initiates the read stream, which begins to accumulate data from all active devices. If the user is ready to receive this data, a call to `oni_read_frame` has to be made. The first call to this function retrieves not a single frame but an entire block of data, sized according to the read buffer. Users can set this buffer size, with larger buffers allowing higher throughput as more data is sent simultaneously. However, this might increase per-frame latency since the entire block must be accumulated before it can be sent, which may contain multiple frames.

When the read buffer is filled, the function returns the first frame in the buffer using zero-copy views into this buffer. Subsequent calls to `oni_read_frame` will return single frames until the frame buffer is depleted. Once the buffer is empty, the library fetches a new block from the read stream.

To write a frame, the user must first create one by calling the `oni_create_frame` function, which returns a frame to which the user can attach the appropriate data. The frame is then passed to the `oni_send_frame` function, which sends the frame to the write stream. In this case, single frames are sent instead of fixed-size blocks. An overview of the data receiving and sending process is shown in Figure 2.7.

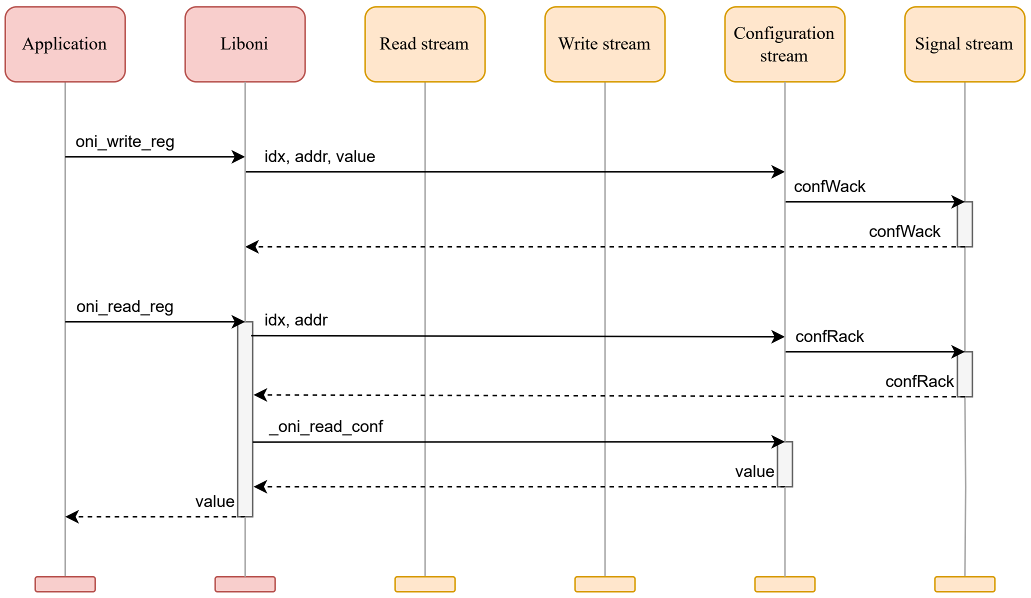


Figure 2.6: Overview of interacting with device registers

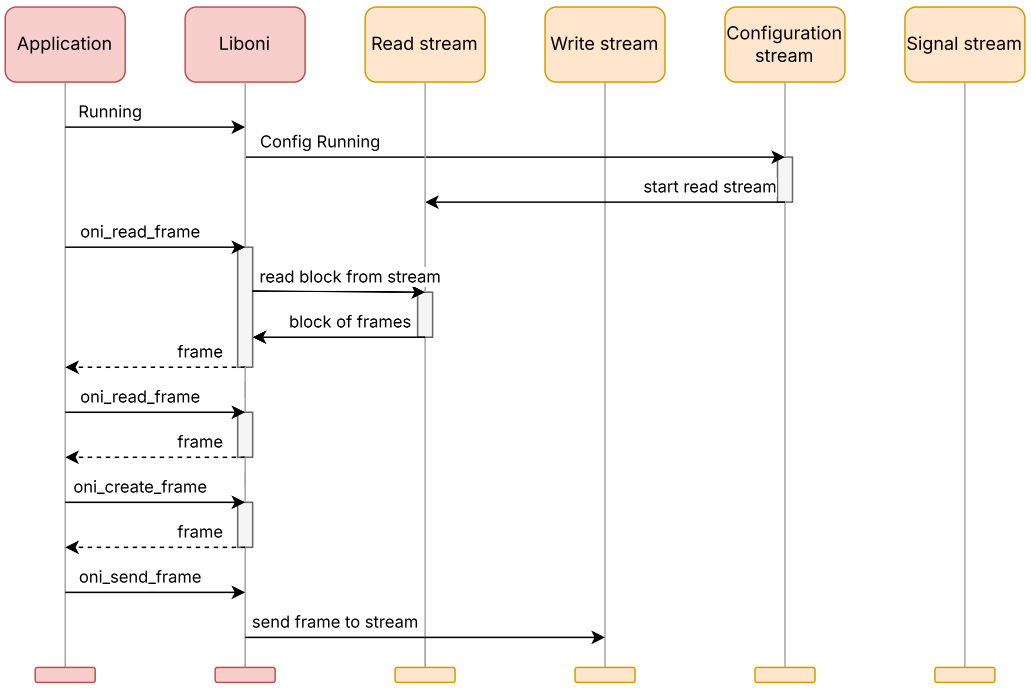


Figure 2.7: Overview of Data Transfer Process in Liboni

2.5.4. Driver development for Liboni

As previously mentioned, Liboni is a hardware-agnostic library that implements the ONI standard, so the underlying hardware can be freely implemented. To use Liboni with a specific hardware platform, a device driver translator (also called onidriver) is required. A device driver translator sits between the public Liboni API and the low-level libraries or kernel drivers handling the actual hardware, taking care of all the implementation details.

To implement a onidriver, a driver context and all functions defined in `onidriver.h` should be implemented. A driver context is a data structure that contains all state information about a particular instance of the driver. The functions defined in `onidriver.h` are the interface between the Liboni API and the device driver translator. They include functions for creating and destroying the driver context, opening and closing the communication channels, reading and writing data and signals, setting and getting device options, reading and writing device registers and handling errors and events. All functions present in`onidriver.h` must be implemented, even if they are not actively used.

2.6. PCIe and Kernel driver

The next part in the ONIX system from Figure 2.1 is the RIFFA Kernel driver and RIFFA FPGA Core. These components enable communication over Peripheral Component Interconnect Express (PCIe). PCIe is a serial data interface used in most modern PC's to connect peripheral devices to the system. [28] Different drivers exist for PCIe but they all share some common elements that are discussed in the following section.

2.6.1. PCIe Bus Organization

A PCIe bus consists of a hierarchy of switches, endpoints and links. A switch is a device that connects multiple PCIe devices and routes packets between them. An endpoint is a device that initiates or terminates data transfers, such as a graphics card or ONIX card. A link is a point-to-point connection between two PCIe devices, consisting of one or more lanes. A lane is a pair of differential signal wires, one for transmitting and one for receiving. Each lane operates at a fixed bit rate, which can be negotiated at the initiation of the link. The PCIe standard defines several generations of speed and width for links, such as PCIe 1.0 x1 (0.250 GB/s per lane), PCIe 2.0 x4 (0.500 GB/s per lane), up to the at moment of writing, latest announced PCIe 7.0 (15.125 GB/s per lane). The throughput of the PCIe is doubled by either taking a next generation or by doubling the amount of PCIe lanes between two links. An example of a PCIe bus is show in Figure 2.8.

The PCIe bus uses a packet-based protocol to transfer data between devices. Each packet consists of a header, a payload and a footer. The header contains information such as the source and destination address, the type and length of the packet and the error detection code. The payload contains the actual data to be transferred, which can be up to 4 KB in size. The footer contains an end-of-packet indicator and an optional error correction code. The PCIe protocol supports four types of packets: memory read and write, which access the main memory; I/O read and write, which access the I/O space; configuration read and write which access the configuration space of PCIe devices; and messages, which carry various control and status information.

2.6.2. Base Address Registers (BAR) Space in PCIe

PCIe devices use Base Address Registers (BARs) to manage memory and I/O resources allocated to them by the system. BARs are a fundamental part of the PCIe configuration space and are essential to facilitate communication between the CPU and PCIe devices. These registers define regions of memory or I/O space that are ”mapped” to the PCIe device, allowing the CPU and other components to read and write data to the device as though it were regular RAM or I/O space.

Each PCIe device can support up to six BARs (BAR0 to BAR5) within its configuration space. These registers hold the base addresses of memory or I/O regions assigned to the device during system initialization. This assignment is handled by the system firmware or operating system. The BARs map device-specific memory or I/O space to the system's address space, ensuring efficient data exchange between the device and the system.

BARs can be categorized into two main types:

- **Memory-Mapped BARs**: These BARs define regions in the system's memory space that are directly accessible by the PCIe device. When configured, these regions can also be accessed

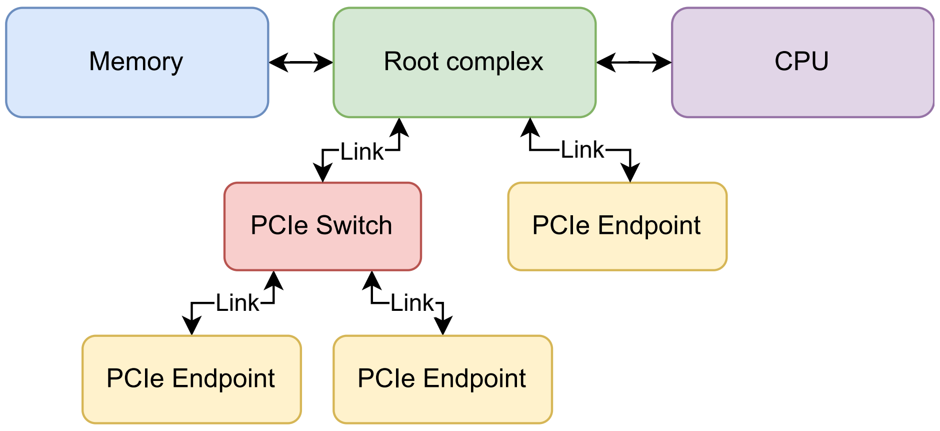


Figure 2.8: Overview of typical PCIe hierarchy

by the CPU using standard memory instructions. The size of the memory-mapped region is specified by the PCIe endpoint device, typically as a power of two, and can be used to store control registers, buffers, or other data structures required by the endpoint device. Memory-mapped BARs also allow for more efficient communication as the device is treated like part of the system's memory.

- **I/O-Mapped BARs**: These BARs define regions in the system's I/O address space. Unlike memory-mapped BARs, I/O-mapped BARs are accessed using specific I/O instructions, such as 'in' and 'out' on x86 systems. I/O mapping was more common in earlier generations of hardware, particularly in legacy devices, because it provided a dedicated address space for peripheral devices that was separate from system memory. However, as memory-mapped I/O (MMIO) became more efficient and flexible, I/O-mapped BARs became less common in modern systems. Today, they are primarily used in legacy systems where backward compatibility is necessary, and certain older devices still rely on this mechanism for communication.

The configuration of BARs involves several critical steps:

1. **Device Discovery and BAR Initialization**: During system boot, the firmware scans the PCIe bus to detect all connected endpoint devices. For each device, the system reads the BARs to determine the amount of address space required by the endpoint device, as each BAR indicates the size and type of memory or I/O space the device needs.

2. **Resource Allocation**: Based on the size and type of address space requested by the BARs, the system allocates appropriate memory or I/O space. Once allocated, the system writes the corresponding base addresses into the BARs, effectively "mapping" the PCIe device into the system's address space. The BARs can specify 32-bit or 64-bit addresses, depending on the device's requirements.

3. **Device Access**: After the BARs are configured, the CPU and other devices can access the memory or I/O regions of the PCIe device by referencing the addresses stored in the BARs. Memory-mapped I/O (MMIO) is commonly used for high-speed communication, where the device's registers and memory appear as part of the system's address space, allowing the CPU to interact with them using regular load/store instructions.

In addition to address allocation, BARs also include flags that indicate the type of address space (32-bit or 64-bit), whether the memory region is prefetchable (i.e., can be cached by the CPU), and other properties. These flags help optimize the interaction between the CPU and the device, ensuring efficient data transfer while minimizing latency.PCIe devices often utilize Direct Memory Access (DMA), where

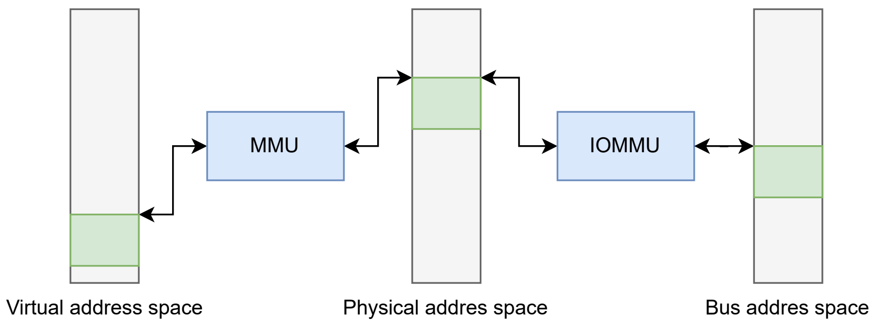


Figure 2.9: Overview of Memory Spaces

the device can read/write to system RAM without CPU involvement, further enhancing performance in high-throughput applications.

2.7. Linux Kernel & Memory Spaces

The Linux operating system is divided into two distinct spaces: **userspace** and **kernel space**. Userspace is where all regular programs running on a PC operate, including user applications and the Liboni library. Programs in this space have limited access to resources to ensure system stability and security.

Kernel space, on the other hand, has access to all hardware on the system. It provides APIs for userspace programs to interact with the hardware. The separation between userspace and kernel space is crucial for both performance and security, ensuring that hardware resources are shared fairly among programs and preventing unauthorized access to sensitive memory regions, such as stored passwords. To maintain this separation, different memory spaces are utilized.

The following memory spaces are important for the implementation of RIFFA. An overview of how these memory spaces relate is shown in Figure 2.9.

User Virtual Addresses: User virtual addresses are the addresses seen by user-space programs. Each program has its own 32-bit or 64-bit address space, depending on the underlying hardware.

Physical Addresses: Physical addresses are used between the processor and the system's memory. This address space is constructed by the system at startup and includes all relevant hardware, such as RAM and devices on the PCIe bus. When a user program allocates memory, it is also allocated in the physical address space. However, since users do not have direct access to this space, a translation is needed, which is handled by the Memory Management Unit (MMU). The MMU contains a page table that translates virtual addresses to physical ones.

Bus Addresses: Bus addresses are used between peripheral devices on a bus, such as PCIe devices and memory. Often, bus addresses are the same as physical addresses. However, some PC architectures have an Input Output Memory Management Unit (IOMMU) between the peripheral bus and the main memory. An IOMMU extends the functionality of the MMU to PCIe devices, mapping device-visible virtual addresses to physical addresses. This allows PCIe devices to access main memory through the Direct Memory Access (DMA) engine. Additionally, an IOMMU provides memory protection from faulty or malicious PCIe devices by preventing unauthorized memory access and supports virtualization by allowing guest operating systems to use PCIe devices not specifically designed for virtualization.

Kernel Logical Addresses: Kernel logical addresses make up the normal address space of the kernel. These addresses map some portion (or possibly all) of main memory and are often treated as physical addresses. On most architectures, logical addresses and their associated physical addresses differ only by a constant offset.

2.7.1. Scatter-Gather and DMA

RIFFA users can pass buffers allocated in user space to the driver, which utilizes them for data transfers using DMA. DMA allows a PCIe device to access the main memory directly without involving the CPU, significantly reducing CPU overhead after the transfer is initiated.

Using DMA, the PCIe device can read from and write to the main memory, facilitating data transfers between the main system and the FPGA. However, for these operations, the PCIe controller on the FPGA needs the memory locations of the user-allocated buffer. Since the user address space and device address space do not match, translations are required. Additionally, user-allocated memory is not necessarily contiguous in physical memory but can be scattered throughout the physical address space.

To address this, the driver creates a scatter-gather list. In a scatter-gather list, the user-allocated buffer is divided into fixed-size pages, often 4KB in size. The virtual addresses are then translated to the corresponding physical addresses. Finally, the kernel finds the bus addresses corresponding to the physical addresses. With the bus addresses, the PCIe engine on the FPGA can read from and write to the user-allocated buffer effectively.

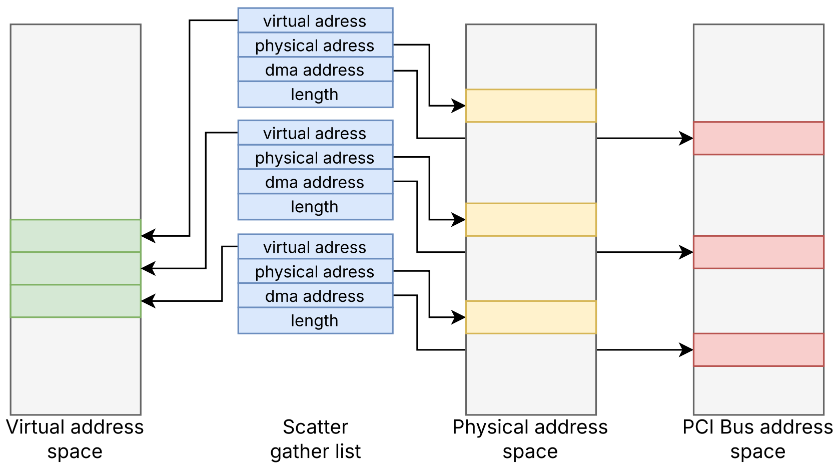


Figure 2.10: Scatter-Gather and DMA Process

2.8. RIFFA

The Reusable Integration Framework for FPGA Accelerators (RIFFA) is an open-source library that facilitates communication between a host CPU and a FPGA via a PCI Express bus [29]. This framework is compatible with both Windows and Linux operating systems, and supports Altera and Xilinx FPGAs. It provides bindings for various programming languages including C/C++, Python, MATLAB, and Java. An overview of the RIFFA architecture is show in Figure 2.11.

The design of RIFFA is based on the concept of communication channels between software threads on the CPU and the logic on the FPGA. Similar to a network socket, a channel must first be opened, can be read and written, and then closed. Furthermore, reads and writes can occur simultaneously if using multiple threads. RIFFA supports up to 12 channels per FPGA, allowing up to 12 different logic blocks to be accessed directly by software threads on the CPU, simultaneously.

On the hardware side, user defined logic can access an interface with independent transmit and receive signals. These signals provide transaction handshaking and a FIFO interface for reading/writing data. RIFFA works directly with the PCIe Endpoint and can run fast enough to saturate the PCIe link.

RIFFA's software primarily consists of two functions: data send and data receive. These functions are accessible through user libraries in various programming languages. The driver supports multiple FPGAs (up to five) per system, enabling users to communicate with FPGA IP cores with minimal code.

Figure 2.12 illustrates how RIFFA functions when receiving data. The logic on the FPGA side can signal at any moment that it wants to send data. In the case of Liboni, this occurs when enough frames are collected to fill the receive buffer. When this happens, a Message Signaled Interrupt (MSI) is sent

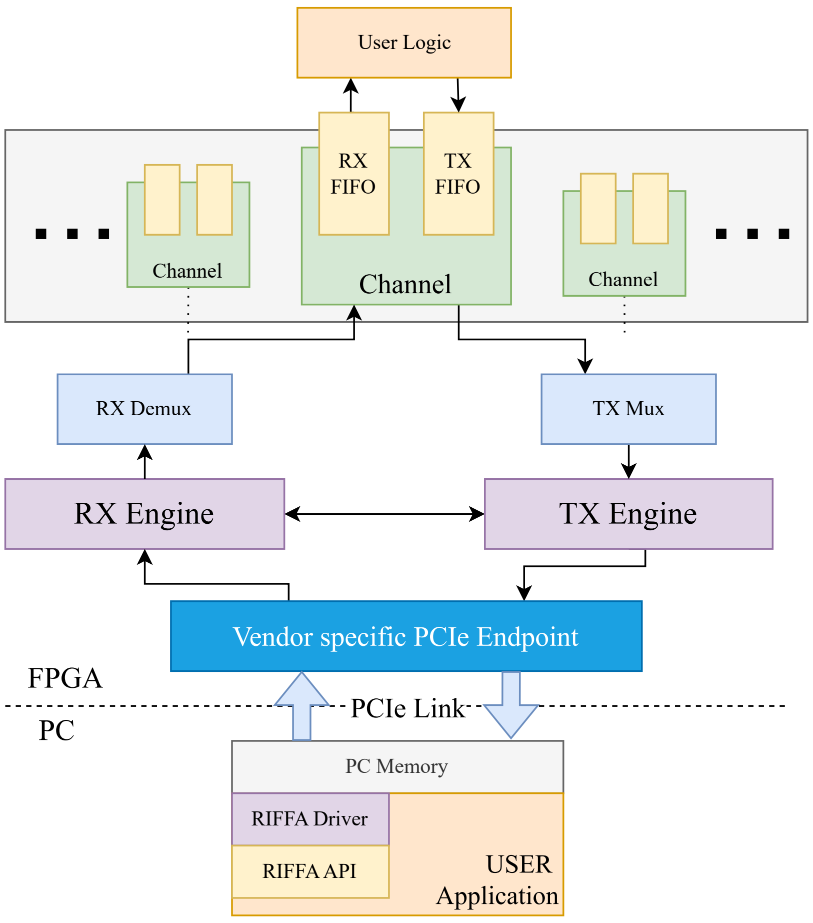


Figure 2.11: Architecture overview of RIFFA

to the kernel driver, which stores the interrupt. When the Liboni library on the PC calls the `fpga_recv` function, the interrupt is handled, and the transfer starts. The RIFFA driver creates a scatter-gather list from the user-allocated memory. This scatter-gather list is then sent to the RIFFA core, which writes into the supplied memory locations. When more memory is needed, a new scatter-gather list is requested, and the process is repeated. Once the RIFFA core has sent all the data supplied by the logic, it sends a new MSI indicating that the transfer is complete. The driver then reads a register on the PCIe device, which holds the total amount of data sent and returns this information to the user.

Figure 2.13 shows how data is sent using RIFFA. Sending data is similar to receiving data from the FPGA, with the main difference being that the transaction is initiated by the driver. The driver sends an interrupt to the logic and subsequently sends the scatter-gather list. The RIFFA core then reads the memory locations from the main memory and transfers the data to an AXI stream to the logic. Finally, the driver reads a register on the PCIe device to confirm the total amount of data sent.

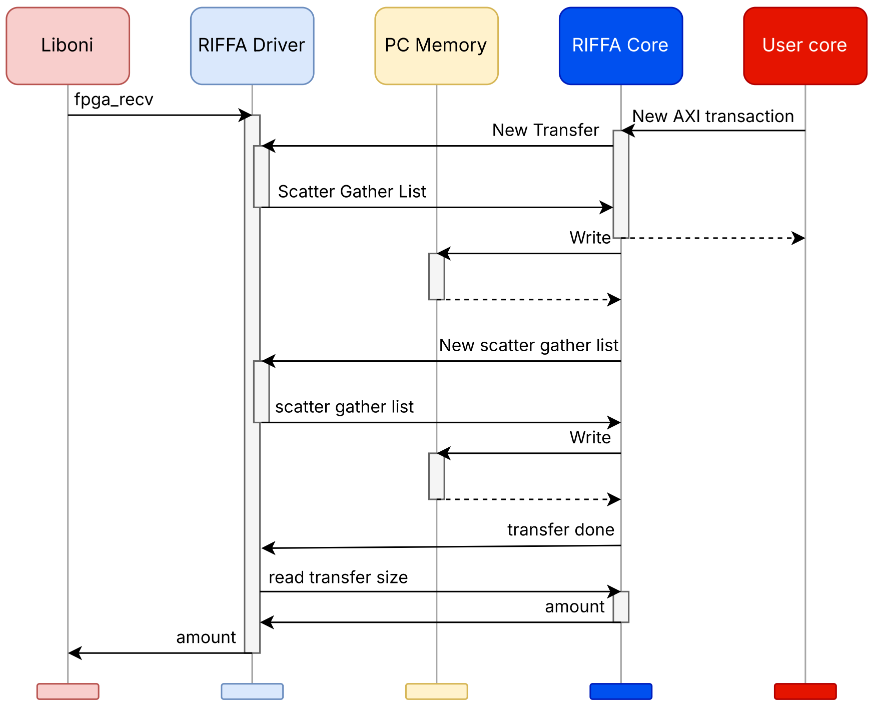


Figure 2.12: RIFFA Data Receiving Process

2.9. Graphics processing unit

Graphics processing units (GPUs) have, as their name suggests, commonly been used for rendering of graphics for games and visually intensive applications. However, they have evolved beyond their traditional role. Today, they are increasingly being used as general-purpose computing devices due to their highly parallel structure, which makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. This concept, known as General-Purpose computing on Graphics Processing Units (GPGPU), allows developers to harness the computational power of the GPU for non-graphical tasks. Languages such as CUDA and OpenCL have been developed to facilitate this kind of programming. GPGPU has found applications in a wide range of fields, including machine learning, scientific computing, and big data analysis, where the ability to process large amounts of data in parallel can significantly reduce computation times. However, programming for GPUs is not without its challenges, as it requires a different programming paradigm and careful consideration of data dependencies and memory usage.

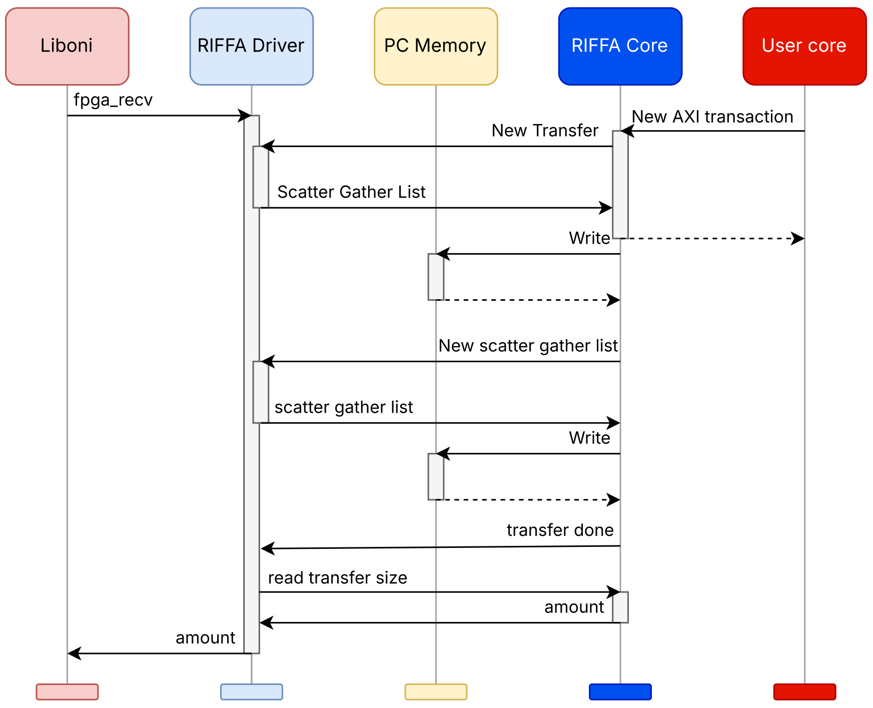


Figure 2.13: RIFFA Data Sending Process

2.9.1. CUDA

The Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows developers to use a CUDA-enabled GPUs for general purpose processing. CUDA provides both a low level API (CUDA Driver API, non single-source) and a higher level API (CUDA Runtime API, single-source), each with its own benefits. The CUDA platform is designed to work with programming languages such as C, C++, and Fortran. This accessibility makes it a popular choice for data scientists and researchers working on complex computational problems. CUDA can greatly improve the speed of GPU-accelerated applications by harnessing the power of the GPU's multi-core architecture. It also provides a comprehensive software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. However, CUDA is proprietary to NVIDIA and therefore, requires an NVIDIA GPU to run. Despite this limitation, the performance gains from using CUDA have led to its wide adoption in the field of high-performance computing.

2.10. GPUDirect RDMA

GPUDirect is a family of technologies designed to accelerate data transfer using PCIe peer-to-peer functionality. Traditionally, when handling data in a computing system with GPUs, all data had to be copied to the main memory first. This ensured that NVIDIA drivers had full control over the data sent to the GPU. However, because of the rise of GPGPU and the need for faster data throughput, NVIDIA introduced GPUDirect. This allows other devices on communication buses to directly read and write data to the memory located on the GPU.

GPUDirect encompasses several technologies:

- **GPUDirect Storage**: Enables GPUs to directly read and write to local or remote storage, such as NVMe.
- **GPUDirect Peer-to-Peer**: Allows communication between GPUs.
- **GPUDirect Video**: Provides an optimized pipeline for frame-based devices.

These technologies are typically available to large hardware vendors creating these types of hardware. However, the final technology, **GPUDirect RDMA**, is openly available and allows any type of device on the PCIe bus to access GPU memory.

GPUDirect RDMA (Remote Direct Memory Access) was introduced in Kepler-class GPUs and CUDA 5.0. Normally, data transfer is handled solely by the GPU kernel driver, which is closed to the user and can only be interacted with using user-space APIs within CUDA. However, GPUDirect provides kernel-level APIs that expose memory locations in the GPU.

To determine if a system supports GPUDirect, several requirements must be met:

- **GPU Support**: Only Tesla and Quadro GPUs, which are server and workstation-grade GPUs, support GPUDirect. GPUs typically found in laptops and desktop computers do not support this feature.
- **Motherboard Support**: The motherboard must support peer-to-peer PCIe communication. Although specified in the PCIe standard, this feature is not universally supported. Motherboards designed for larger workstations and servers are more likely to support it.
- **IOMMU Restrictions**: GPUDirect imposes restrictions on the IOMMU (Input-Output Memory Management Unit). Specifically, the IOMMU must be configured to allow direct access to GPU memory by devices on the PCIe bus, bypassing the standard IOMMU translation process. In some systems, the IOMMU's memory address translation and protection mechanisms can interfere with direct data transfers between devices and the GPU. To enable GPUDirect RDMA, the IOMMU may need to be disabled or configured in a pass-through mode to ensure that memory access is not blocked or altered by the IOMMU. This reduces the overhead associated with memory translations but can also limit certain system security features typically provided by the IOMMU, such as isolation of device memory accesses.
- **Operating-System Support**: Currently, only Linux distributions support GPUDirect.

2.11. Related work

Electrophysiology experiments in neuroscientific research are inherently complex, and scientists often lack an extensive background in computer science or electrical engineering. Consequently, a significant amount of research has been dedicated to developing platforms that simplify the setup of these experiments. This section discusses various aspects of this research.
First, electrophysiology (ephys) systems other than the ONIX hardware are explored. Following this, the software used to interface with ephys systems is examined. Additionally, research on accelerating communication between FPGAs and GPUs is reviewed, as this is relevant to the thesis.

2.11.1. Ephys Systems

Electrophysiology (ephys) systems have been extensively studied in the context of animal experiments, with a variety of tools available for neural recording and analysis. One of the commonly used systems is the Intan RHD recording system, a modular and cost-effective electrophysiology data acquisition platform centered around the Intan RHD family of microchips [14]. This system includes RHD recording headstages capable of digitizing signals from 16, 32, 64 or 128 electrodes. These digital signals are then transmitted to a recording controller, which sends the data from one or more headstages to a PC via USB.

Intan also sells their chips for custom hardware development. An example of this is the first system built by the creator of ONIX, called the Open Ephys Acquisition Board [32]. Open Ephys Acquisition Board is designed to be affordable, transparent, and flexible, supporting up to 512 channels. Similar to the RHD controller, it connects to a host PC using USB.

Another notable system is the open-source, open-loop 1024-channel recording system called Willow [18]. Willow also utilizes Intan chips for front-end amplification and filtering but distinguishes itself by sending data directly to a solid-state drive, thereby eliminating the need for a separate computer to acquire data. Additionally, a copy of the data can be transmitted over Ethernet for real-time visualization. Although Willow is optimized for high channel count reliability, it lacks the modularity and software extensibility of Open Ephys.

Mukherjee, Wachutka, and Katz [23] introduced an affordable, scalable, and open-source system for use with awake, behaving rodents, emphasizing its cost-effectiveness and scalability. Their system demonstrates that a Raspberry Pi can successfully interface with up to four RHD200-based headstages. Additionally, it can drive optogenetic perturbations and actuators using the Raspberry Pi's GPIO pins. Their results indicate that relatively simple hardware can be effective when the experimental constraints are not demanding. Finally, Topalovic et al. [37] showcased a wearable bidirectional closed-loop neuromodulation system (Neuro-stack) that records single-neuron and local field potential activity during both stationary and ambulatory behavior in humans. The Neuro-stack system includes a TPU for performing inference with a machine learning model, demonstrating the potential for GPU-like hardware in closed-loop neuroscience applications. The mentioned ephys systems are summarized in Table 2.2.

System	Open-Source	Max Channels	Connection Type	Special Feature
Intan RHD [14]	No	128	USB	None
Open Ephys [32]	Yes	512	USB	None
Willow [18]	Yes	1024	Ethernet	Direct-to-drive
Mukherjee et al. [23]	Yes	128	USB	Cost effective
Neuro-stack [37]	No	128	Direct onboard connection	TPU Accelerator

Table 2.2: Taxonomy of Ephys Systems

2.11.2. Ephys Software

In electrophysiology research, specialized software frameworks are essential for managing complex experiments, especially those involving real-time closed-loop applications. These tools enable precise control, dynamic perturbations, and efficient data processing, significantly advancing our understanding of neural dynamics.

One example is the Real-Time eXperiment Interface (RTXI), which provides low-latency, hard real-time capabilities for closed-loop experiments. Built on a custom Linux kernel, RTXI supports flexible protocol implementation in a wide range of experimental settings. Despite its strengths, is is only available for commercial data acquisition (DAQ) hardware, which can limit accessibility [27].

The Open Ephys GUI, developed alongside the Open Ephys hardware platform, offers a modular, plugin-based interface for data acquisition, processing, and visualization. Although optimized for Open Ephys hardware, the GUI also supports interoperability with other systems, enhancing its versatility. Integration with tools like Bonsai further expands its functionality, enabling seamless management of diverse experimental setups [32].

Bonsai, a visual programming framework, builds on this versatility by facilitating advanced workflows. Designed for real-time event-based processing, Bonsai is well-suited for tasks such as behavioral experiment control, video-electrophysiology synchronization, and closed-loop applications. Its modular architecture make it particularly effective for computationally intensive tasks [20].

Another notable framework is Falcon, a multi-threaded C++ platform designed for high-performance closed-loop experiments. Falcon supports the creation of complex processing graphs with sub-millisecond latencies, enabling applications like dynamic clamp, real-time spike detection, and neural oscillation analysis. Its flexibility and compatibility with various hardware systems make it a valuable tool for computationally demanding experiments [8]. Table 2.3 provides a concise comparison of these tools.

2.11.3. Direct FPGA-GPU Communication

The implementations of direct communication between FPGAs and GPUs via the PCIe bus, as explored in various studies, illustrate a range of innovative approaches aimed at enhancing performance and efficiency in heterogeneous computing systems. Direct communication over PCIe has been applied in scenarios involving Software-Defined Radio [35] or Radar Signal Processing [30].

Bittner and Ruf [4] introduced an innovative approach where existing DMA engines on GPUs are used for DMA operations to an FPGA. By mapping the FPGA's memory into the process's virtual address space, they could register it as DMA-safe, enabling faster data transfers from the GPU to the FPGA. This method increased data throughput for large transfers, though it faced challenges with re-

Table 2.3: Taxonomy of Ephys Software Suites

	RTXI[27]	Open Ephys GUI[32]	Falcon[8]	Bonsai[20]
Open Source	Yes	Yes	Yes	Yes
Identifying Features	Hard real-time capabilities; closed-loop control; custom Linux kernel	Plugin based architecture; flexible data acquisition and visualization	Multithreaded architecture; sub-millisecond latency; real-time detection	Visual programming; modular workflows
Supported OS	Linux	Linux, Windows	Linux	Windows

verse transfers, particularly with larger data sizes, where performance dropped significantly. Despite some limitations, the approach effectively reduced data transfer latency.

In another approach, Thoma et al. developed an open-source framework named FPGA2, designed to facilitate DMA between GPUs and FPGAs. This framework utilizes a Xilinx IP core the open-source GPU drivers, Nouveau [25] and the gdev [17] an open-source implementation of CUDA runtime. By pinning GPU memory and obtaining its bus address, FPGA2 performs DMA in a manner similar to GPUDirect RDMA. Although it requires an additional memory copy within the GPU, the impact on performance is minimal. For smaller transfers, FPGA2's performance is approximately double that of traditional host-mediated transfers, though it becomes comparable for larger data sizes. The framework's reliance on open-source tools makes it suitable for consumer-grade GPUs.

In more recent work, Kasai and Osana [15] presented two distinct implementations of DMA transfer between FPGA and GPU without the involvement of host memory. Both methods involved mapping the memory on either the FPGA or GPU to their respective BAR (Base Address Register) space, allowing the partner's DMA controller to directly read or write from/to the BAR. The first method maps the FPGA's memory to the FPGA BAR using a PCIe-AXI bridge, enabling GPU DMA access. This method is very similar as the approach taken by Bittner and Ruf [4]. While this approach is straightforward and avoids driver-level programming, it results in significant performance degradation compared to methods that involve host memory. The second method maps the GPU memory to the GPU BAR via the GPUDirect API, allowing FPGA DMA access. This approach nearly saturates the PCIe bandwidth, achieving performance that was 1.6 times better than the host memory-based method. However, transferring data from the GPU to the FPGA failed. As future work, it is mentioned that RIFFA might be used, to improve performance.

In table 2.4 the three different implementations are compared. A comparison of the maximum achieved throughput in each case reveals significant differences in performance. Given the varying number of PCIe lanes and PCIe generations used in each solution, a more meaningful comparison can be made by examining the utilization of the PCIe lanes.

The FPGA^2 solution performs well in this regard, achieving a utilization of 81% and 75% for FPGA-to-GPU and GPU-to-FPGA communication, respectively. However, this solution does not scale effectively as it is implemented for only one lane. The solution proposed by Bittner & Ruf shows decent performance, with an 80% utilization in the GPU-to-FPGA direction, but significantly lower performance in the FPGA-to-GPU direction. Notably, it is the only solution that operates under Windows, making it the only viable option if that is a constraint.[4]

Kasai & Osana's solution achieves the highest utilization in the FPGA-to-GPU direction, with 81%. [16] However, it does not successfully implement GPU-to-FPGA communication. A solution to this limitation is not mentioned in the paper.

Table 2.4: Comparison of Direct FPGA-GPU Communication in Previous Work

Property	**Bittner & Ruf [4]**	**FPGA^2 [36]**	**Kasai & Osana [16]**
Operating System	Windows	Linux	Linux
DMA Master	GPU	FPGA	FPGA
FPGA Vendor	Xilinx	Xilinx	Xilinx
FPGA Model	Virtex 6	Virtex 5	Alveo U50
FPGA IP Stack	Custom	Custom	Vendor
FPGA Driver	Custom	Custom	XDMA
FPGA Programming	HDL	HDL	HDL
GPU Vendor	NVIDIA	NVIDIA	NVIDIA
GPU Model	GeForce GTX580	GeForce 8400GS	Quadro RTX 4000
GPU Driver	Original	Nouveau	Original
GPU Programming	CUDA	gdev	CUDA
Effective PCIe Lanes	8	1	4
PCIe Generation	1.0	1.0	3.0
Maximal Throughput (FPGA to GPU)	514 MB/s	203 MB/s	32,500 MB/s
Maximal Throughput (GPU to FPGA)	1.6 GB/s	189 MB/s	N/A

3 Design and Implementation

This chapter outlines the design changes and implementations needed to enhance data throughput and reduce latency in the ONIX system, when integrating a GPU. It begins with a description of a straightforward approach using high-level APIs for closed-loop GPU integration, then progresses through the required system modifications for improving troughput and latency.

Key updates are discussed across the FPGA core, Linux kernel driver, and Liboni library, each essential to facilitate efficient data transfer. In the FPGA core, new data paths are added and channel widths increased to optimize PCIe bandwidth. Driver modifications include adapting the RIFFA driver to support P2P data exchange with pre- and non-pre-pinned memory configurations. Lastly, the Liboni library is extended to accommodate new user options for managing data streams and direct GPU communication.

3.1. Base Case

Before examining GPU integration into the ONIX system, it is important to highlight the two primary modes in which the ONIX system is used and discuss how optimization strategies differ for each. These modes are as follows:

Open-Loop Data Acquisition: In this mode, data acquisition occurs independently of any influence on the ongoing experiment. This setup allows for real-time data processing as it arrives, which can enable the researcher to obtain preliminary results quickly. However, in open-loop experiments, maximizing data throughput and minimizing latency are not critical requirements. Instead, the focus is on ensuring that sufficient memory and computational resources are available for post-processing tasks.

Closed-Loop Experiments: In closed-loop experiments, acquired data directly influences the future course of the experiment. This mode demands fast data processing to enable real-time feedback during the experiment. Key metrics for optimization include minimizing the time required to transfer, process, and return data to the experiment. For this reason, the focus of this thesis will be on optimizing the system for closed-loop scenarios.

For both open and closed loop experiments, the simplest method to leverage GPU resources is by utilizing high-level APIs provided by the GPU vendor, such as NVIDIA's CUDA or OpenCL. Consider an experiment where data is collected from multiple ephys devices, but only one device's data requires GPU acceleration. In this scenario, each frame from the ONIX card must be processed individually. When a that should be processed by the GPU is detected, the data is copied to a new buffer. Once sufficient data has been accumulated, it is transferred to the GPU, where a CUDA kernel can execute the necessary algorithm. The results are then copied back to the user program, and a new ONI frame is created and sent back to the ONIX device to generate an output signal. The following steps are thus involved, which are also depicted in Figure 3.1.

1. The first call to `oni_read_frame` internally returns a block of data from the FPGA and the first frame in this block is returned to the user.

2. This block of data can then be read out frame by frame. If a frame contains data that should be sent to the GPU, the data is copied in a secondary buffer.
3. The buffer containing data for the GPU is sent using `cudaMemCpy`.
4. A Cuda kernel is run and the result is stored in memory.
5. The result is copied back to the main memory using `cudaMemCpy` again.
6. The result is written into the data part of an ONI frame.
7. The ONI frame is sent back to the FPGA using `oni_write_frame`.

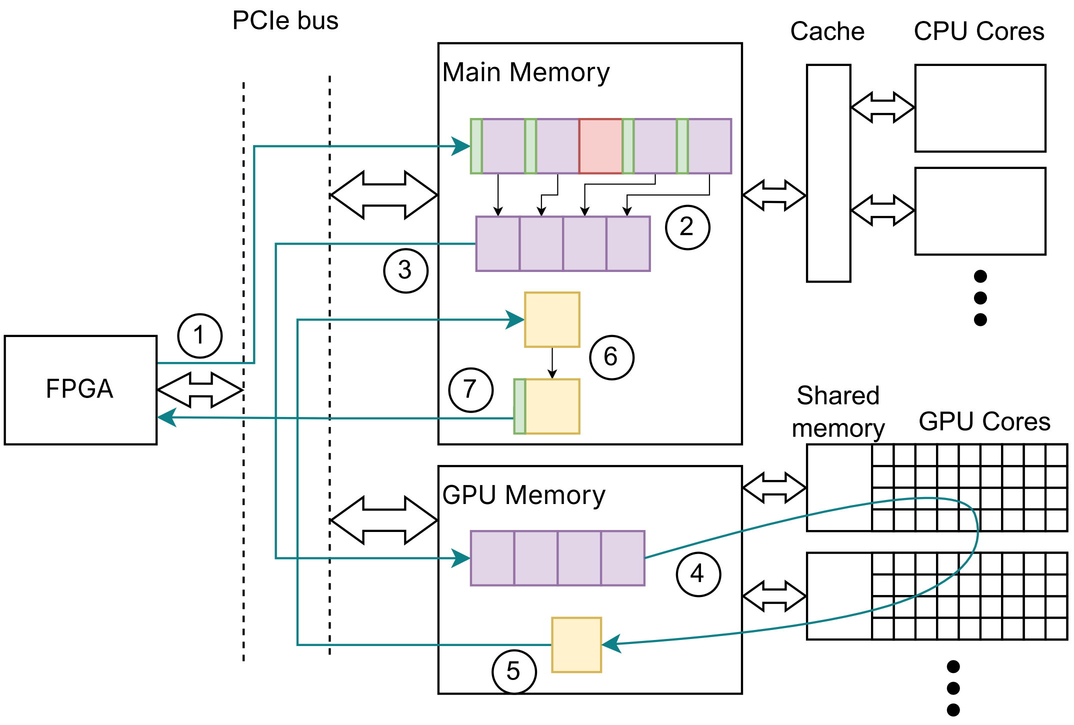


Figure 3.1: Steps involved in sending data to the GPU from the current ONIX system

3.2. Reducing Transfer Steps

One strategy to reduce the overhead associated with data transfer between the FPGA and GPU is to decrease the processing workload on the CPU by shifting more responsibilities to the FPGA. Currently, during steps 2 and 6 (as described in the previous section), the CPU must inspect each frame transferred between the FPGA and GPU to determine which data should be processed by the CPU and which by the GPU. To eliminate this overhead, the data flow can be restructured so that, instead of multiplexing all data into a single stream, two separate data streams are created: one designated for the CPU and another for the GPU. This restructuring modifies the data transfer steps between the FPGA and GPU as follows. The steps are also depicted in Figure 3.2.

1. A mechanism is implemented on the FPGA to allow the user to specify which data is buffered for the CPU and which for the GPU.
2. The user initiates a library call that transfers a block of data from the FPGA into the main memory.

3. The buffer containing data for the GPU is transferred using `cudaMemcpy`.
4. A CUDA kernel is executed, with results stored in memory.
5. The result is copied back to main memory using `cudaMemcpy`.
6. The ONI frame is written back to the FPGA using a GPU-specific write function.
7. Data is send to the destination device.

Implementing these changes requires modifications at multiple levels of the technology stack. First, within the FPGA logic, a method must be developed to separate data intended for the CPU and GPU, as detailed in Section 3.4. Additionally, the `liboni` library must be updated to support these changes, as discussed in Section 3.6.

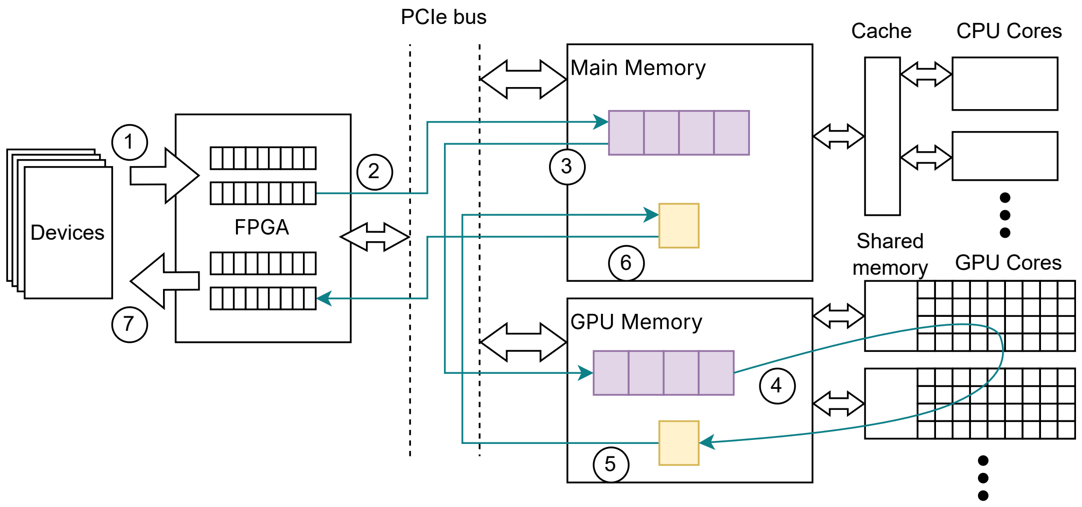


Figure 3.2: Closed loop data transfer from FPGA to GPU with reduced steps when compared with base case

3.3. Direct PCIe transfer

As was discussed in Chapter 2, communication between a FPGA and a GPU can be made faster by directly sending data between them on the PCIe bus instead on using main memory. To design a system where data is directly communicated over the PCIe bus two major design decisions have to be made. The first one is which PCIe endpoint behaves as the master, either the GPU or FPGA. If the GPU works as the master a approach similar to Bittner and Ruf [4] has to be taken. While this approach would work on Windows OS, it has the main disadvantage that the way that direct communication was achieved was more of a hack than a supported feature of NVIDIA GPUs. Another disadvantage is that memory of the GPU needs to be mapped to the bus. This does not work with RIFFA as it is designed for the FPGA to work as a bus master and thus an entirely new communication layer for the ONIX has to be designed. For this reason it is preferred that the FPGA functions as bus master in the proposed design.

The second design decision is how to write data to and from the FPGA. Two approaches can be taken in this. The first is by directly addressing memory on the FPGA. This means that a large chunk of memory is pinned to the PCIe bus and the user can decide where to read and write data from. This is the approach taken by Thoma, Dassatti, and Molla [36] and Kasai and Osana [15]. While this approach is very flexible since the user can read and write anywhere in the memory space of the FPGA, it adds complexity in the design as this all needs to be managed in a memory safe way. Another approach would be the one RIFFA takes. Instead of giving the user the option to read and write to a block of memory, they can read and write from fixed channels which from the user's perspective can be regarded as large buffers. Although no literature exists that implies this approach would work when reading and writing data from the GPU memory, it is by far the easiest to implement since ONIX has been designed around RIFFA. Thus it will be tested if a modified version of RIFFA which is able to directly read and write to and from the GPU memory can increase throughput and latency compared to the base case explained above. On a high level, the system will have to perform the following tasks, which are also depicted in Figure 3.3.

1. There needs to be a way for the user to select which data is buffered for the CPU and which data is buffered for the GPU on the FPGA
2. The user makes a library call that transfers a block of data from the FPGA in to the GPU Memory
3. A Cuda kernel is run and the result is stored in memory.
4. Another library call is made that transfers the data back to the FPGA
5. The user shoudld be able to configure which device the data is eventually written to.

The kernel module which handles the data transfers to and from the FPGA needs to be modified such that the data can be transferred to and form the GPU Memory, these changes are discussed in Section 3.5. Finally the ONI library needs additional API calls to make it possible for the user to initiate and configure the transfers, these changes are discussed in Section 3.6.

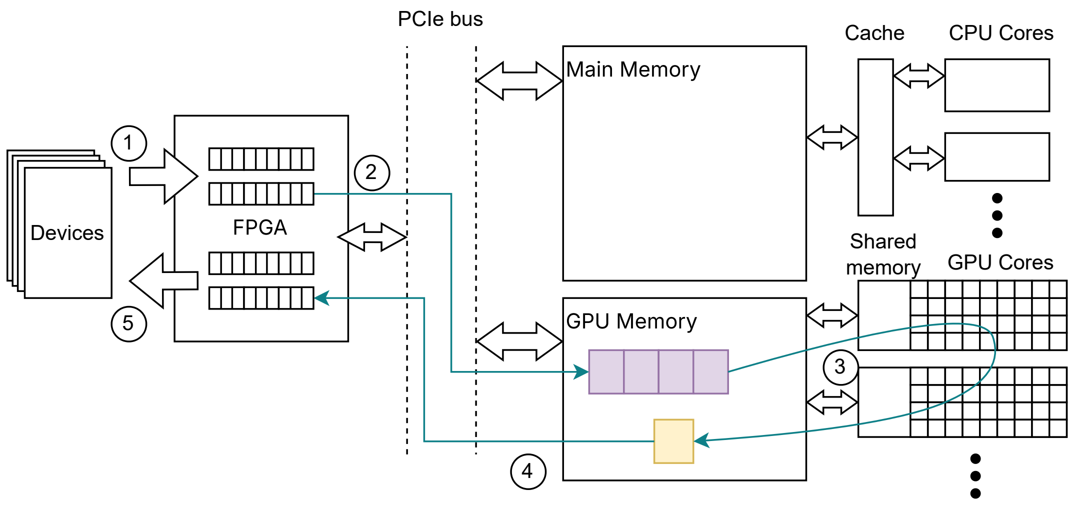


Figure 3.3: Steps involved in sending data directly over PCIe to the GPU

3.4. ONIX FPGA-Core Modification

If direct communication is to be established between the FPGA and GPU, then there needs to be a way to separate the data meant for CPU and GPU inside the FPGA. To do this separation, an extra data stream on the FPGA is required, which collects all data which should be sent to the GPU. This setup allows one data stream to remain versatile for CPU processing, while the second stream can be tailored for high-performance, closed-loop operations using the GPU.

To achieve this, RIFFA should have two additional channels, one for reading GPU data and one for writing. The RIFFA core, currently in use, can be extended to support extra channels, each capable of sending and receiving data independently. To these channels, a GPU write stream and GPU read stream are connected, which operate exactly the same as the existing read and write stream. The only difference between the original data paths and the added ones is the GPU data input/output block. The original data input/output blocks multiplex and demultiplex data from all ephys devices into a single data stream. The GPU data input/output should be configurable, allowing a single device to write and read to the input and output, respectively. This ensures that no two ephys devices write data simultaneously, thereby preventing data mix-up. Additionally, the data width of the input and output paths can be increased. In the original design, these paths are 16 bits wide, which, at a clock speed of 250 MHz from the PCIe lines, offers a maximum throughput of 500 MB/s.

Furthermore, introducing a second read data stream means that there are two components that read from and write to the DDR3 memory independently. To accommodate this, the DDR3 memory should be partitioned into two separate regions, allowing each data stream to access its respective partition without interference. Since only one device can access the memory controller at a time, an interconnect is added to manage arbitration between the two data streams. A design proposition of the new FPGA firmware is shown in Figure 3.4.

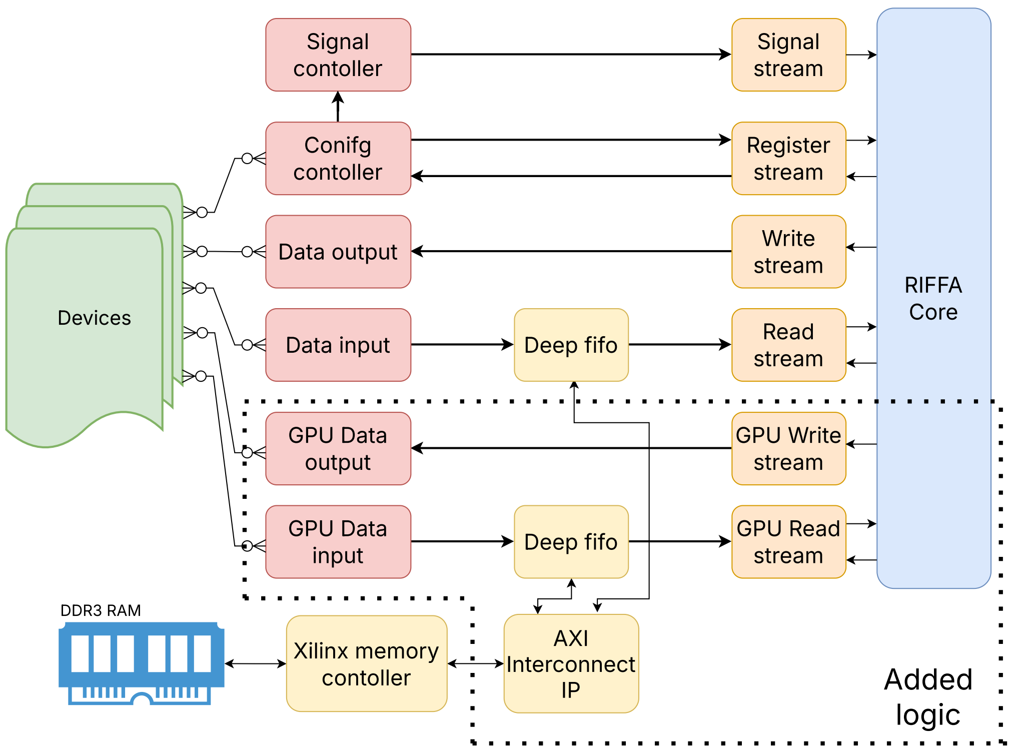


Figure 3.4: Overview of the updated ONIX FPGA core design with and additional read and write data for GPU data

3.5. Linux RIFFA-Kernel Driver Modification

Building on the concepts discussed in Section 2.8 regarding RIFFA and in Section 2.10 on GPUDirect, this section outlines the necessary modifications to the RIFFA driver to enable peer-to-peer communication between the FPGA and GPU. The communication mechanism proposed is similar to that used between the main memory and FPGA, where the kernel creates a scatter-gather structure that the FPGA uses to read and write data. This procedure is explained in Section 2.7.1.

A key difference between sending data to a location in main memory compared to GPU memory is that device addresses are not automatically exposed on the PCIe bus. Instead, at startup, the GPU requests a portion of the PCIe bus address space using its Base Address Register (BAR). The amount of address space requested by the GPU varies between GPU models. For example, the Quadro P620 GPU, which is used in later experiments, has BAR1 allocated for read and write operations with a size of 256 MB, which means that only a portion of the GPU's 2 GB memory can be mapped to the PCIe bus at any one time.

To map allocated memory to the PCIe bus, the `nvidia_p2p_get_pages` function is used. When a user allocates memory within the user program, the Nvidia driver returns a virtual address. This virtual address is then passed to the `nvidia_p2p_get_pages` function, which maps the corresponding memory pages to the PCIe memory address space within the BAR. This process, known as pinning the memory, generates a page-table structure containing the addresses and sizes of the pinned pages. Memory pinning is a critical yet time-consuming operation, as noted in the GPUDirect documentation [26].

To enable the FPGA to read and write to these memory addresses on the PCIe bus, a final address translation is required. Due to the presence of a Memory Management Unit (MMU) between the PCIe bus and CPU, the I/O addresses of PCIe resources used in P2P transactions differ from the physical addresses used by the CPU. The `nvidia_p2p_dma_map_pages` function performs this translation, ensuring that the physical addresses align with those on the PCIe bus, thereby making them accessible to other devices on the PCIe bus. Figure 3.5 provides an overview of how these functions and memory relate to each other. Given the time-intensive nature of memory pinning, optimizing this process is crucial. Two strategies for pinning are considered:

- **Without Prepinning**: This method is analogous to the original RIFFA functionality. If a buffer exceeds the maximum size of the scatter-gather structure, which is 200 pages, it is divided into multiple smaller scatter-gather structures that are sequentially pinned and sent to the FPGA. This approach allows the transfer of data to a buffer significantly larger than the GPU's maximum pinning size. The maximum scatter-gather size is 200 times the size of a GPU page (64KB), resulting in up to 12.8 MB of data being pinned to the BAR at one time. However, the constant pinning and unpinning for large transfers introduces additional overhead.
- **With Prepinning**: This method mitigates the overhead associated with frequent pinning and unpinning by pre-pinning the read or write buffer before data transfers begin. This approach is particularly effective in the Liboni library, where the same buffers are reused throughout the experiment. The method requires additional kernel calls, `rdma_pin` and `rdma_unpin`, to pin the buffer once before the experiment starts, ensuring that it remains mapped to the PCIe bus for the duration of the experiment. However, the amount of data that can be pinned is limited and dependent on the specific GPU in use.

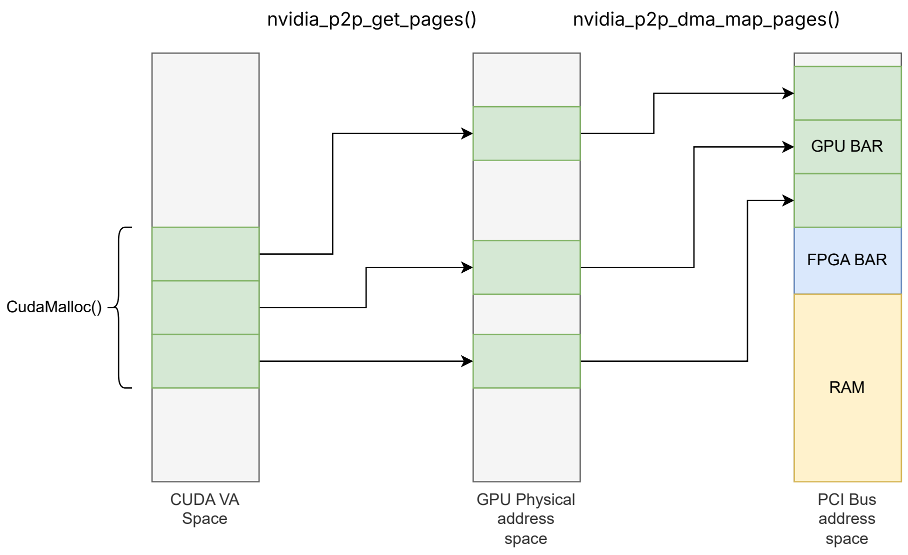


Figure 3.5: Memory Mapping in GPUDirect

3.5.1. Wrapper Kernel Module

A challenge when using GPUDirect is that the functions provided by the NVIDIA driver are licensed under a proprietary license, whereas all other APIs provided by the Linux kernel are licensed under the

GNU General Public License (GPL). The GPL includes a ”copyleft” provision, which mandates that any derivative work based on GPL-licensed code must also be distributed under the same license. Because of this copyleft requirement, the Linux kernel prohibits mixing proprietary code with GPL-licensed code within the kernel. Consequently, GPUDirect functions cannot be directly integrated into a custom kernel module that calls linux kernel functions.

To overcome this issue, a secondary wrapper module was developed. This wrapper module encapsulates the necessary NVIDIA functions, allowing them to be called by the driver without violating the provision. The reason why the wrapper can call the NVIDIA driver functions is that the driver does not make calls to any other GPL-licensed functions. Thus, it does not mix proprietary function calls and GPL function calls. This approach ensures compliance with licensing requirements while still allowing the use of GPUDirect functionality.

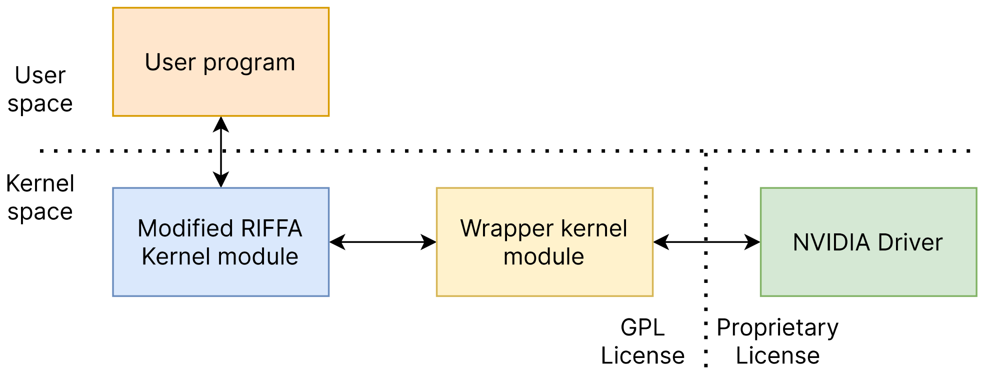


Figure 3.6: Overview of how wrapper kernel module connects to NVIDIA Driver

3.6. Liboni Library Update

Changing the liboni library should be done with some care. A big appeal of the ONIX system is that it is standardized and easy to set up. Therefore, changes made to the library and API shouldn't affect the usage of ONIX too much. First off, there needs to be a way for the user to configure whether a device sends or receives data through the default data stream or the new GPU data stream. To enable this, devices that support GPU computation should include a register that toggles this behavior via the `oni_set_opt()` function.

The next problem is that the GPU data stream is not designed to send individual packets, the existing API functions `oni_read_frame()` and `oni_write_frame()` are incompatible with this stream. The only way that custom driver based functionality can be built into the API is using `oni_driver_set_opt` and `oni_driver_get_opt`. To implement all optimizations that have been discussed in this chapter there needs to be a way to pin memory, get the handle return by the pinning and upinning of memory; there also needs to be a way to set the read and write buffer size; finally the user should be able to initiate read and write actions to the driver. An overview of the options that should be added to the library is shown in Table 3.1.

3.7. Conclusion

This chapter addressed the question of how the current ONIX design can be updated to enable communication to a GPU. First, a a solution without any modifications to the current design is shown (Figure 3.1). Utilizing high-level APIs, such as NVIDIA's CUDA, which allows for closed-loop experiments with GPU acceleration.

Next, a more optimized version of the system is discussed which still send data trough host memory but does so more optimized by reducing the steps involved.

Get/Set	Option Name	Input/Output	Explanation
Set	ONI_DRV_RD_SZ	Size	Set the size of the Read buffer
Set	ONI_DRV_WR_SZ	Size	Set the size of the write buffer
Set	ONI_DRV_PIN	Memory address	Pin the buffer at the given memory location
Set	ONI_DRV_UNPIN	Memory address	Unpin the buffer at the given memory location
Set	ONI_DRV_READ	Memory address	Write a block of data to the given memory address to the FPGA
Get	ONI_DRV_RD_SZ	Size	Returns the current set size of read buffer
Get	ONI_DRV_WR_SZ	Size	Returns the current set size of write buffer

Table 3.1: Library call options added to the RDMA driver to add custom functionality

To implement PCIe P2P communication, modifications to the ONIX system are necessary. Data meant for the CPU or GPU needs to be separated inside the FPGA before transmitting. This involves updating the FPGA image to support multiple data channels. Furthermore, the data width of these channels is increased to fully utilize the PCIe bandwidth.

Additionally, enabling peer-to-peer communication between the FPGA and GPU requires the development of a new Linux kernel driver. This driver is based of the original RIFFA driver and can interface with the RIFFA core on the FPGA. Two modes of sending are added, one with memory prepinning and the other without memory prepinning.

The Liboni library, which interfaces with the ONIX system, also needs to be updated to accommodate these changes. This includes adding options for driver-specific functions such as memory pinning, setting buffer sizes, and initiating read and write actions directly to the driver. These updates will allow for the integration of advanced GPU computation capabilities while maintaining the flexibility and ease of use that the ONIX system is known for.

4 Performance Evaluation

In this chapter, we investigate the performance of GPU integration within the Open Ephys system. The hardware changes proposed in the previous chapter are implemented and the impact on transfer time and troughput are measured.

The first section describes the experimental setup and methods used for data transfer, where we implement a streamlined FPGA core to simulate Open Ephys data handling. This core supports up to 2 GB/s throughput, allowing an accurate assessment of latency and bandwidth without extensive system alterations.

In the second section, we present and compare results from the original system and the updated system with firmware modifications, focusing on the improvements in data transfer latency and throughput.

The final section examines the impact of GPUDirect on data transfer rates with and without prepinned memory across different payload sizes, providing insight into the conditions that yield optimal performance for high-speed data transfers between the FPGA and GPU.

4.1. Experimental Setup

The testing was carried out on a Precision 7920 Tower workstation, which was selected for its capability to support peer-to-peer communication on the PCIe bus. To make sure the translation of physical addresses and PCI bus addresses performs correctly, Intel's VT-x virtualization technology was disabled. The system specifications and software configurations used in the experiments are detailed in Table 4.1.

The experiments were designed to evaluate the performance limits of data transfer between the FPGA and GPU within the ONIX system. Since closed loop performance is main interest both the FPGA to GPU and GPU to FPGA transfer is evaluated. All the configurations discussed in chapter 3 are tested which are:

- **Original system**: Logic on the FPGA is the official Open Ephys bit-steam, original RIFFA kernel driver and Liboni library is used. Data is geathered in main memory and then send to GPU using cuda functions
- **Updated system**: Updated FPGA Logic, Kernel driver and Liboni library. Data is first send to the main memory then to GPU using CUDA Functions
- **GPUDirect without prepinning**: Updated FPGA Logic, Kernel driver and Liboni library. Data is send directly from the FPGA to GPU without prepinning the memory
- **GPUDirect with prepinning**: Updated FPGA Logic, Kernel driver and Liboni library. Data is send directly from the FPGA to GPU with prepinning the memory

primary goal was to test how the system handles varying data sizes during transfer operations. The metrics of interest are transfer time and troughput. Which are defined as the following:

- **Transfer time**: This is taken as the time from the moment the user issues a command to start the data transfer up until the user gets an indication that all the data has been successfully transferred
- **Throughput**: The total amount of data that is transferred during an experiment divided by the total time of the experiment.

First, the average transfer time for the data to get from the FPGA to the GPU is measured. The data transfer size is progressively increased, starting from a minimum of 64 bytes, and doubling each time up to a maximum of 16.7 MB. For each datasize the transfer is repeated until a total of 8.6 GB of data is transferred. This was chosen as a trade off where result accuracy did not really improve while experiment time would increase significantly by transferring more data. The total time taken for these transfers was measured using C++'s built-in `chrono` library. Using that the average, Standard Deviation (SD) and maximum was calculated for each transfer size.

Component	**Description**
CPU	Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz (40 cores)
System	Precision 7920 Tower
RAM	6x 16GiB DIMM DDR4 Synchronous @ 2933 MHz
GPU	NVIDIA Quadro® P620
FPGA	AMD Kintex-7 XC7K160T
Operating System	Ubuntu 20.04.6 LTS (kernel 5.4.0-190-generic, 64-bit)
C++ Compiler	GCC 9.4.0
FPGA Synthesiser	Vivado
GPU Drivers	Nvidia 535.183.01 (Linux)
CUDA	CUDA 12.2

Table 4.1: System Hardware and Software Configuration

4.2. FPGA testing logic

For the purpous of this experiment, instead of implementing the FPGA logic defined in section 3.4, logic specific for testing the system is designed. The problem with propped system for testing purpouses is that devices have to be used to generate data while, for this experiment the interest lays in the behaviour of the connection between FPGA and GPU. Therefore this test logic is implemented which is able to send arbitrary data back and forth between the FPGA and GPU, but which should perform exaclty the same as the designed system for section 3.4.

The core includes two inputs and two outputs. The data source and sink are used to generate and receive data at the maximum throughput that the FPGA can handle. The FPGA operates at a clock frequency of 250 MHz, provided by the PCIe bus. The data source outputs an incrementing 64-bit value at every rising clock edge, resulting in an maximum output rate of 2 GB/s. The sinks is non-blocking and capable of always receiving data also at a rate of 2 GB/s

To simulate the FPGA's operation when data is stored on the DRAM, both input and output streams are incorporated. Data is written to the DRAM using the write stream and can subsequently be read using the read stream. This feature is particularly useful for testing applications where the FPGA's output data must adhere to a specific type or format. Users can format the data and send it to the FPGA, which is then redirected to the output. The RIFFA core is not changed and operates the same as in the original ONIX design. A schematic overview of the new core is given in Figure 4.1. Since in both this test design and the design specified in Section 3.4 the DRAM is used as a buffer and all other components between the DRAM and RIFFA core are the same, this test logic should be a good representation of the performance of the designed system.

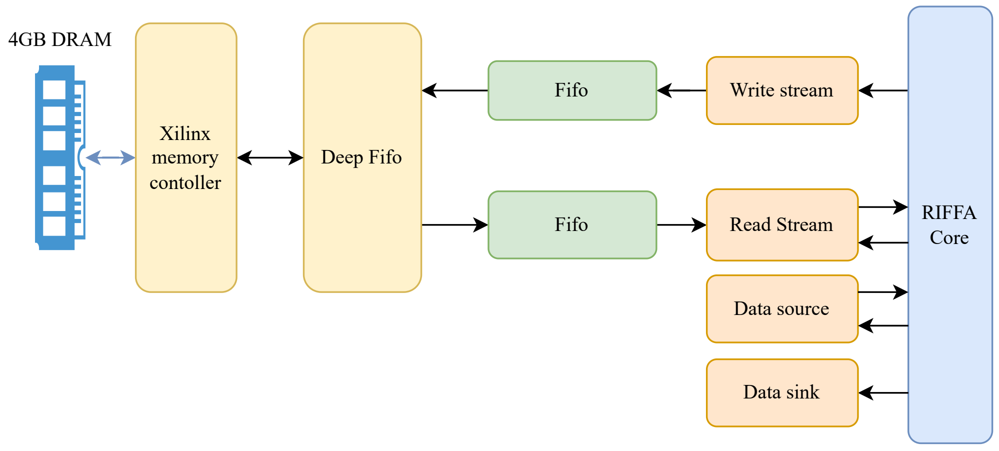


Figure 4.1: Simplified FPGA-Core Test Bed

4.3. Performance of Original System

The initial performance evaluation of the Open Ephys system was conducted using all the original software parts. The original FPGA logic contains a test device capable of generating artificial data to fully saturate the data path and receiving data at the same rate. The test device was configured to operate at the highest possible throughput. The frame size generated by the test device was set to match the block read size, ensuring that each call to `oni_read_frame()` resulted in a single driver call that returned the frame. irst, buffers are allocated on both the host and GPU sides using `cudaMallocHost()` and `cudaMalloc()`. Once the data is correctly received, it is copied to the GPU using CUDA's memory-copy function. Finally, a call to `cudaStreamSynchronize()` blocks the execution of the next transfer until all data has been properly written to the GPU. The code in Listing 4.1 provides an overview of the measurement process used to evaluate system performance.

Listing 4.1: Reading and processing frames

```
cudaMallocHost(&host_buffer, transfer_size);
cudaMalloc(&device_buffer, transfer_size);
for(int i=0; i < repeats; i++){
    oni_frame_t *frame = NULL;
    rc = oni_read_frame(ctx, &frame);
    cudaMemcpy(device_buffer, frame->data, block_read_size, cudaMemcpyHostToDevice);
    cudaStreamSynchronize(cudaStreamDefault);
    total_read += frame->data_sz;
    oni_destroy_frame(frame);
}
```

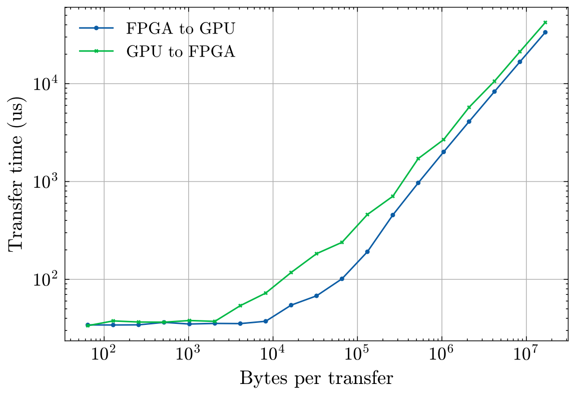
4.3.1. Results

The experimental results are presented in the following figures. Figure 4.2a illustrates the average latency as a function of the number of bytes transferred per operation. A key observation is that latency linearly increases at around 10 kB. This indicates that the throughput has plateaued. Calculating this throughput shows that for transfers from the FPGA to the host a maximum of approximately 500 MB/s can be reached. In contrast, the throughput for transfers from the host to the FPGA is lower, saturating at 350 MB/s.

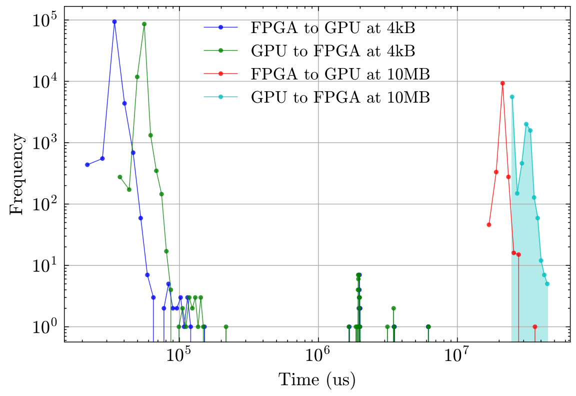
Figure 4.2b is a frequency plot of the transfer time for two transfer sizes, highlighting significant differences in latency, particularly for smaller transfers. Some number of the graph are shown in in Table 4.2 and indicate an average SD of 47 µs and 52 µs for FPGA-to-GPU and GPU-to-FPGA transfers, respectively, which is similar order of size as the average transfer time. Additionally, the maximum transfer time observed is exceptionally high, exceeding 100 times the average. For larger transfer sizes, the time for FPGA-to-GPU transfers is noticeably lower compared to GPU-to-FPGA transfers.

Transfer Type	**Throughput (MB/s)**	**Transfer Time (µs)**		
		Average	**SD**	**Max**
FPGA to host, 4 kB	116	34	47.3	6190
Host to FPGA, 4 kB	76	51	52	6210
FPGA to host, 10 MB	499	20900	644	3730
Host to FPGA, 10 MB	382	27700	4150	45500

Table 4.2: Performance metrics for FPGA-to-host and host-to-FPGA transfers.



(a) Average transfer time trough host memory before changes to ONIX



(b) Transfer time before changes to ONIX

Figure 4.2: Performance of the ONIX system without any upgrades

4.3.2. Conclusion

The performance evaluation of the original ONIX system reveals clear limitations, particularly in terms of throughput and latency. The maximum achievable throughput is constrained to approximately 500 MB/s for FPGA-to-GPU transfers, this is due to the fact that the original FPGA logic can only ouput 16 bits per clock cycle at 250MHz. The constraint means that the current data path in the logic FPGA is a bottleneck. To fully utilize the available PCIe bus and GPU resources, improvements to the FPGA data path are necessary.

The observed transfer time differences, especially for smaller transfers, are likely due to the shared access to main memory across the system and the lack of prioritization in memory-access scheduling. Implementing GPUDirect, which bypasses the main memory, could potentially address these overhead. Additionally, the poorer performance observed in GPU-to-FPGA transfers can be attributed to the additional memory allocations required in the current system. When generating a frame which is send to the FPGA new memory is always allocated, while when getting data from the FPGA preallocated buffer are used. Eliminating these unnecessary allocations could further enhance system performance.

Overall, the results underscore the need for firmware enhancements and architectural modifications to optimize the ONIX system for high-performance GPU computation.

4.4. Performance of Updated system without GPUDirect

This section the updated system is tested and compared to the original system. The test will use the FPGA core shown in Figure 2.3, which is able to read and write data at 2GB/s. Listing 4.2 demonstrates how data is transferred from the FPGA to the GPU using the newly implemented driver and library functions but without GPUDirect. First, buffers are allocated on both the host and GPU sides using `cudaMallocHost()` and `cudaMalloc()`. The entire transfer is similar as the previous experiment except that the new liboni library options are used to set the transfer size and initiate data transfer from the FPGA to the host. The GPU to FPGA transfer occurs in a similar manner.

Listing 4.2: Reading and processing frames with the new system

```
cudaMallocHost(&host_buffer, transfer_size);
cudaMalloc(&device_buffer, transfer_size);
oni_set_driver_opt(ctx, ONI_DRV_RD_SZ, transfer_size, 4);

for(int i=0; i < repeats; i++){
    rc = oni_set_driver_opt(ctx, ONI_DRV_READ, host_buffer, 4);
    if (rc != 0)
        error_exit(rc, ”Error reading data\n”);
    cudaMemcpy(device_buffer, host_buffer, transfer_size, cudaMemcpyHostToDevice);
    cudaStreamSynchronize(cudaStreamDefault);
}
```

4.4.1. Results

The results of the experiments are illustrated in Figure 4.3a at a later point in the chapter for a better comparison, but are summarized in Table 4.3. When compared to the orignal system, for smaller transfer sizes, the transfer times remain comparable; however, for larger transfers, the time is much lower. Calculating the throughput using the transfer time shows that the new system plateaus at approximately 1600 MB/s, a significant improvement over the original system. Comparing the performance of the original and modified systems when transferring larger data, there is a substantial decrease in transfer time. The time for FPGA to GPU transfers went from 21000 us to 7658 us. Similarly, GPU to FPGA transfers saw an decrease from 28000 to 7084 us, reflecting an almost 400% improvement.

Figure 4.3b depicts the spread of transfer times after the system modifications. The results indicate that the changes have not significantly impacted the Standard Deviation (SD) and maximum tranfer time. This shows that the underlying contention for system resources remains similar to the original setup.

Transfer Type	**Throughput (MB/s)**	**Transfer Time (µs)**		
		Average	**SD**	**Max**
FPGA to GPU, 4kB	120	34	43	7686
GPU to FPGA, 4kB	119	33	44	14974
FPGA to GPU, 10MB	1379	7658	381	14719
GPU to FPGA, 10MB	1501	7084	426	13731

Table 4.3: Performance metrics for different types of FPGA-to-GPU and GPU-to-FPGA transfers in the modified system.

4.4.2. Conclusion

The experimental results demonstrate significant improvements in system performance following the modifications. The latency for both 10MB FPGA-to-GPU and GPU-to-FPGA transfers has substantially decreased for large transfer sizes. Specifically, FPGA-to-GPU latency improved almost 300%. Similarly, GPU-to-FPGA transfers saw an improvement of nearly a 400%.

Despite the substantial gains in average transfer time, the modifications did not significantly impact the deviation and maximum transfer time; this can be expected since system resource contention remains similar to the previous configuration. However, these enhancements clearly demonstrate that the system modifications have effectively increased the data-transfer rates between the FPGA and GPU, making the system more suitable for high-performance applications and testing of the PCIe bus.

For smaller transfers, the system changes had less of an impact. The time stayed the same for FGPA to GPU transfers but slightly increased for transfers from GPU back to the FPGA.

4.5. Performance of GPUDirect without Prepinning

The GPUDirect functionality is evaluated by first testing it without utilizing prepinning. The evaluation procedure closely mirrors the previous tests, where data was transferred through the main memory. A sample of the code is shown in Listing 4.3. Initially, a buffer is allocated in the GPU memory using `cudaMalloc`, and the data is subsequently transferred to this buffer using the `oni_set_driver_opt` function and the newly implemented `ONI_DRV_READ` option, which handles the data transfer. The transfer process is timed and recorded.

Listing 4.3: Reading data from ONIX using GPUDirect without prepinning

```
cudaMalloc(&device_buffer, transfer_size);
oni_set_driver_opt(ctx, ONI_DRV_RD_SZ, transfer_size, 4);

for(int i=0; i < repeats; i++){
        rc = oni_set_driver_opt(ctx, ONI_DRV_READ, device_buffer, 8);
        if (rc != 0)
        error_exit(rc, "Error reading data\n");
}
```

4.5.1. Results

The results obtained from implementing GPUDirect are shown below. Figure 4.3c presents the average transfer time of the new system using GPUDirect, while Figure 4.3d illustrates the frequency plot for transfer times. The performance results for the two transfer sizes are summarized in Table 4.4.

When comparing the results of GPUDirect with the previous system improvements, several observations can be made. For small transfers (4kB), the transfer time using GPUDirect is significantly higher than that achieved while sending data through main memory. Specifically, the time for FPGA-to-GPU and GPU-to-FPGA transfers at 4kB increased to 155 us and 152 us, respectively. This represents an increase in latency of almost 400% compared to not using GPUDirect. Also the SD and maximum transfer time increased.

For larger transfers (10MB), the transfer time using GPUDirect is comparable to that of sending data through host memory, with transfer sizes around 1MB and beyond surpassing the latency of transfer through main memory. The Standard Deviation (SD) and maximum time metrics slightly improve with GPUDirect but not significantly compared to the average time decrease, suggesting that it does not enhance the real-time performance of the system and probably still contends with shared system resources.

Transfer Type	**Throughput (MB/s)**	**Transfer Time (µs)**		
		Average	**SD**	**Max**
FPGA to GPU, 4kB	31	155	106	15525
GPU to FPGA, 4kB	28	152	107	19766
FPGA to GPU, 10MB	1485	7086	344	15748
GPU to FPGA, 10MB	1598	6517	353	18132

Table 4.4: Performance metrics for different types of FPGA-to-GPU and GPU-to-FPGA transfers using GPUDirect.

4.5.2. Conclusion

The results indicate that GPUDirect without prepinning performs poorly for small transfers, exhibiting higher latency compared to the previous system improvements. Additionally, there is no improvement in the deviation and maximum transfer time, suggesting that GPUDirect does not enhance the real-time performance of the system. The poor performance is likely due to the overhead associated with memory pinning, which will be examined in more detail in section 4.7. Although GPUDirect achieves slightly higher throughput and lower latency for larger transfers, the bad performance at smaller transfers outweigh these benefits. Consequently, GPUDirect without prepinning does not offer a meaningful advantage for the ONIX and may not be worth the constraints it introduces, given its limitations.

4.6. Performance of GPUDirect with Prepinning

The final evaluation that is performed is the GPUDirect with pinning. The evaluation code from a user's perspective looks very similar to the code without pinning, shown in Listing 4.4. The only difference is that the allocated buffer must be pinned before transfer and unpinned after. One notable drawback of this approach is the fact that the amount that can be pinned is defined by how much can be pinned to the BAR space of the PCIe bus. This should be 256 MB according to the hardware specs of the Quadro P600. However, for unknown reason it was found that pinning fails for buffer larger than roughly 16 MB; therefore, the tests do not go further than that.

Listing 4.4: Reading data from ONIX using GPUDirect with prepinning

```
cudaMalloc(&device_buffer, transfer_size);
oni_set_driver_opt(ctx, ONI_DRV_RD_SZ, transfer_size, 4);
rc = oni_set_driver_opt(ctx, ONI_DRV_PIN, device_buffer, 8);
for(int i=0; i < repeats; i++){
        rc = oni_set_driver_opt(ctx, ONI_DRV_READ, device_buffer, 8);
        if (rc != 0)
        error_exit(rc, "Error reading data\n");
}
rc = oni_set_driver_opt(ctx, ONI_DRV_UNPIN, device_buffer, 8);
```

4.6.1. Results

The results of implementing the new method are shown below. Figure 4.3e presents the average latency for the new system, the results for sending data through host memory are also plotted for comparison. Figure 4.3f give the frequency of the transfer times. The performance metrics for various transfers are summarized in Table 4.5.

When comparing the results with the GPUDirect pinning method to the system without GPUDirect, several observations can be made. The transfer time for small transfers (4kB) using the new method shows a significant improvement compared to the transfers achieved without GPUDirect. For FPGA to GPU and GPU to FPGA transfers at 4kB, the transfer time decreased approximately by 30%.

For larger transfers (10MB), the transfer time using the new method is also slightly lower than the results through host memory, achieving an improvement of 14%, indicating that the new method has faster transfers for both small and large transfer sizes. Compared to the transfer trough host memory the SD and maximum time only slightly improves but not significantly.

Transfer Type	Throughput (MB/s)	Transfer Time (µs)		
		Average	SD	Max
FPGA to GPU, 4kB	161	25	40	8010
GPU to FPGA, 4kB	168	24	37	8006
FPGA to GPU, 10MB	1572	6677	267	12686
GPU to FPGA, 10MB	1705	6121	279	12215

Table 4.5: Performance metrics for different types of FPGA to GPU and GPU to FPGA transfers.

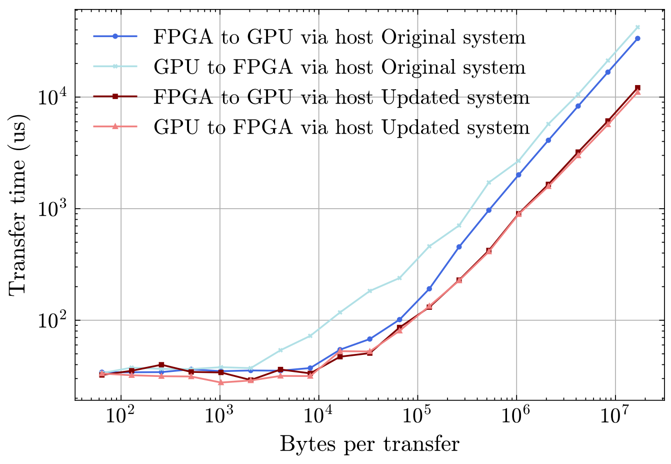
4.6.2. Conclusion

The implementation of GPUDirect with prepinning demonstrates significant benefits over the unpinned version. The primary improvement stems from the reduced overhead of pinning, as this step is performed only once per buffer rather than per transfer. This optimization results in substantial improvements in transfer times and throughput, particularly for smaller transfers, where the overhead of the pinning operations is most pronounced.

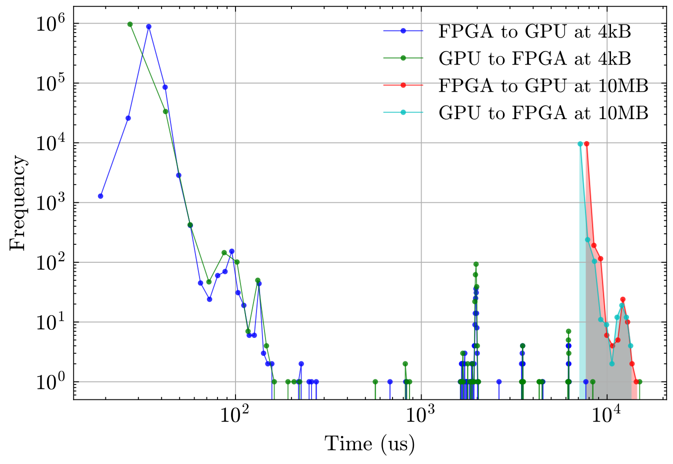
Also when compared to sending data trough host memory the pinned GPUDirect version improves transfer time and throughput. For example, time for 4kB transfers decreased by approximately 30% compared to transfers using host memory.

For larger transfers, while the benefits are less pronounced, prepinning still provides an improvement. The 14% throughput increase for 10MB transfers suggests that even with larger data sizes, avoiding the host memory pathway contributes to measurable performance gains. However, the impact of prepinning on variability, such as jitter and maximum latency, remains minimal. This indicates that prepinning alone cannot address timing inconsistencies in the system.

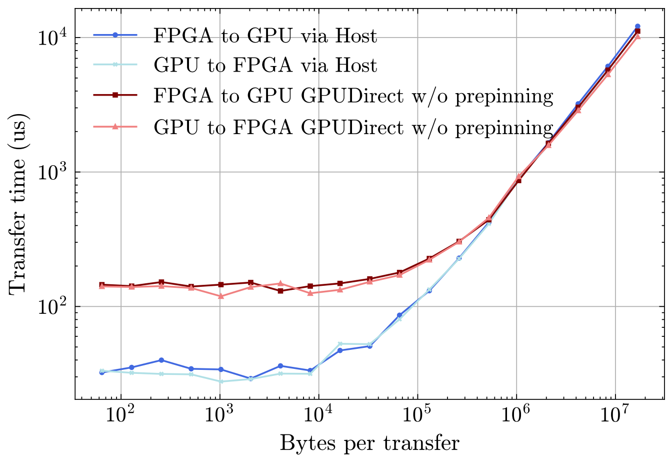
These results highlight the importance of leveraging prepinning for GPUDirect in applications requiring frequent, high-speed transfers, particularly for smaller data sizes. However, further optimizations will be needed to address non-deterministic timing behavior and ensure reliable performance for real-time systems.



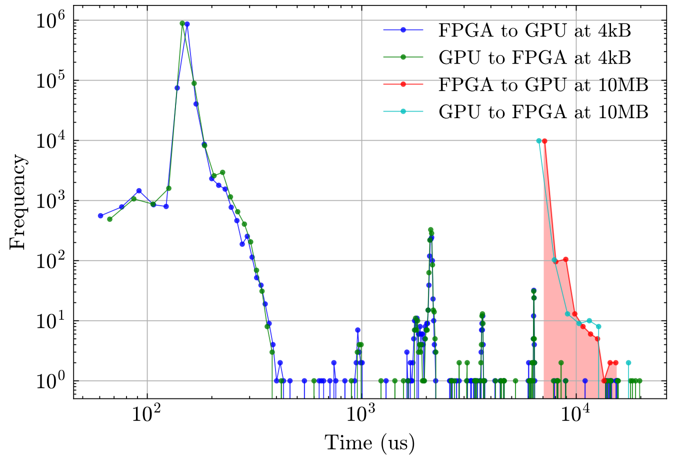
(a) Average transfer time trough host memory after changes to ONIX



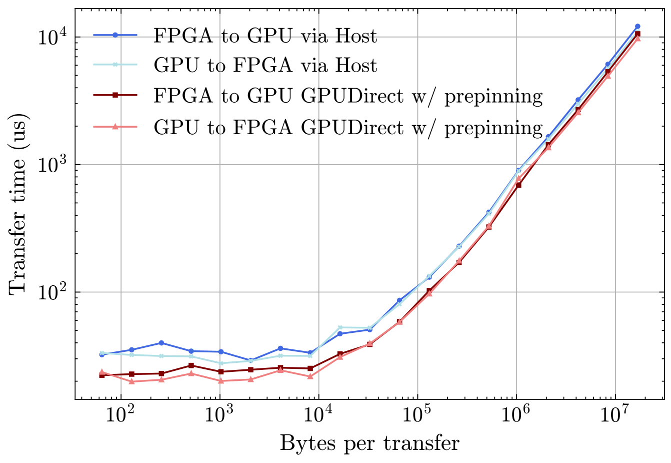
(b) Transfer time after changes to ONIX



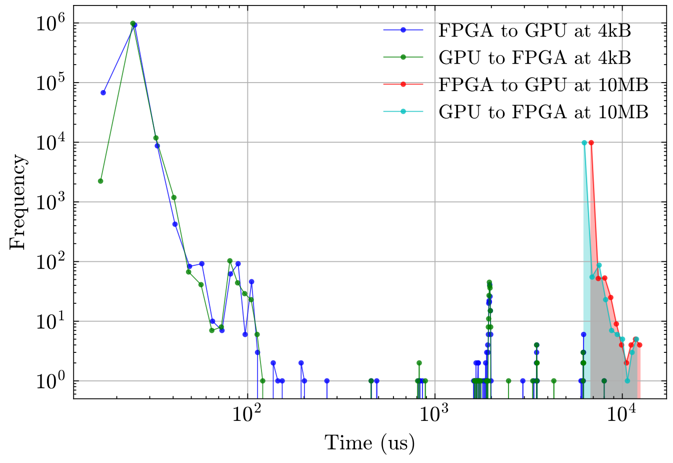
(c) Average transfer time using GPUDirect without prepinning



(d) Transfer time using GPUDirect without prepinning



(e) Average transfer time using GPUDirect with prepinning



(f) Transfer time using GPUDirect with prepinning

Figure 4.3: Comparison of transfer times and throughputs for various configurations.

4.7. Profiling kernel driver

In Section 4.5 the assumption was made that the having to pin the memory for every transfer was the cause of the slower transfers. To verify this a profiling of the modified RIFFA kernel dirver has to be done. Profiling the kernel also helps with identifying the major overheads in the kernel driver. Understanding these overheads is crucial for making informed decisions about potential optimization targets. Profiling also ensures that the kernel behaves as expected. The profiling was performed using the `printk` function, which logs messages to the system log, and the `ktime_get_ns` function, which provides highly accurate system timing. These functions were combined in a macro called `PRINT_TIME`, which was incorporated at compile time. The overhead caused by printing was measured by performing two prints and calculating the time difference between them.

4.7.1. Results

The profiling results are shown in Figure 4.4. Each function in the new kernel module that handles data transfer was timed. The profiling revealed that, for the default and pre-pinned GPUDirect transfers, most of the time was spent waiting for the FPGA to send interrupts indicating it had completed the data transfer. All other parts were negligible and therefore categorized under the ”Other” label. This was not the case for the unpinned GPUDirect transfers, where each transfer involved pinning and unpinning the data. This operation caused a significant overhead, which explains the higher latency for these particular transfers, as seen in Figure 4.3c. A profiling of 10MB transfers was also done but the graph is omitted as it did not provide any additional information. It showed that the driver spends almost all its run time waiting on interrupts.

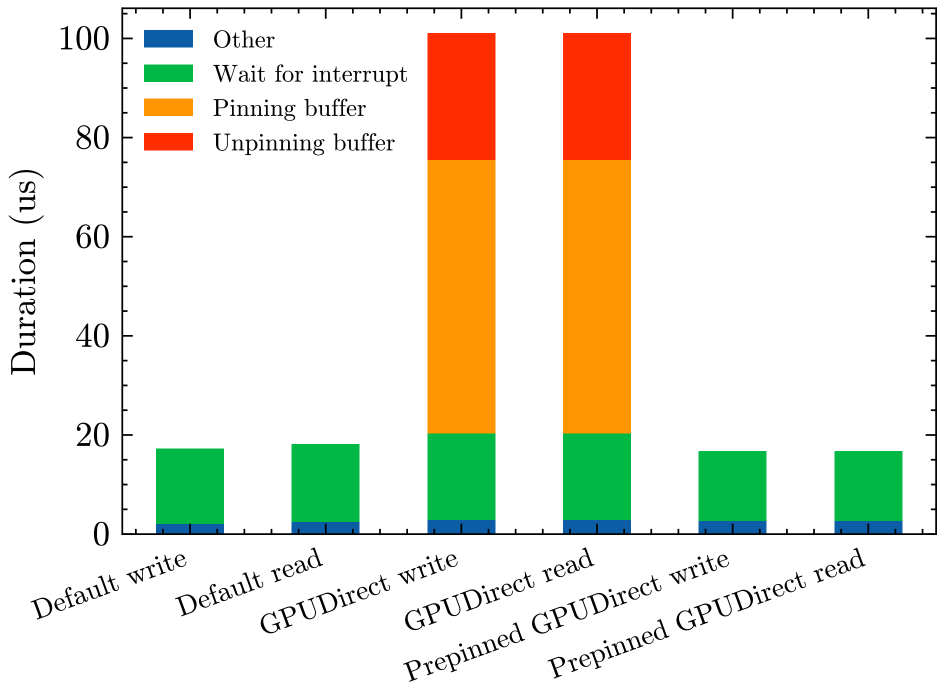


Figure 4.4: Profile of kernel module with 4KB transfer

4.7.2. Conclusion

The profiling results indicate that there is limited scope for optimizing the kernel driver itself. The primary way to significantly reduce transfer time is by decreasing the wait time for interrupts. This could potentially be achieved by speeding up the PCIe bus, either by increasing the lane count or upgrading the generation type. This is however not a posibilty with the current ONIX hardware. Profiling also confirms that the higher latency in the GPUDirect transfers where the memory is not prepinned is mainly due to the time-consuming pinning operations that has to be performed for each transfer.

4.8. Conclusion of performance evaluation

First off, the GPUDirect version without prepinning does not bring any benefits and should not be used by users due to the large overhead associated with pinning each transfer. In contrast, the pinned GPUDirect version clearly demonstrates improvements in both transfertime and throughput for data transfers between the FPGA and GPU. These enhancements are particularly notable for smaller transfers, where the overhead associated with transferring data through the host memory is relatively larger. In these cases, GPUDirect significantly reduces transfer time, resulting in a throughput increase of approximately 30% for 4kB transfers. This improvement is attributed to the elimination of the main memory as an intermediary, which reduces the overhead that disproportionately affects smaller transfers.

For larger transfers, the benefits of pinned GPUDirect are more modest. Although there is an improvement in throughput and latency, the impact is less pronounced. This is because the PCIe channel between the GPU and host is already much faster compared to the FPGA-to-host path, so the relative speedup from bypassing the host memory is smaller. The throughput for 10MB transfers showed a more modest increase of about 14%, reflecting the diminishing returns of GPUDirect as transfer sizes increase.

Despite these throughput and transfer time improvements, GPUDirect did not result in better jitter or maximum latency performance. The lack of improvement in these metrics indicates that GPUDirect alone is insufficient to address the variability and worst-case timing behavior in the system. This suggests that other strategies, such as implementing a real-time Linux kernel, may be necessary to achieve consistent low-latency performance and reduce jitter.

Overall, while GPUDirect offers substantial benefits for improving throughput and latency, especially for smaller transfers, additional measures will be required to enhance real-time performance in high-demand applications. From profiling the kernel driver it was concluded that the only way to further speed up the transfers is by improving the physical PCIe connections.

5 Application Scenarios

The objective of this chapter is to find out in which application scenarios GPU's and GPUDirect provide improved performance. Up to this point, the measurements conducted have focused on stress testing the system with data transfers, to determine the maximum feasible throughput. However, to understand the differences in throughput between the CPU and GPU, it is essential to consider scenarios where multiple processes are executed concurrently. These processes include data transfer from the ONIX FPGA card, computation, and response transmission. Ideally, these operations should occur simultaneously to maximize throughput. Consequently, this chapter evaluates the system in scenarios where both data transfers and processing are performed concurrently.

Three different configurations are compared in this evaluation:

- CPU-only configuration.
- GPU computation with data transferred through host memory.
- GPU computation with pre-pinned GPUDirect memory.

In practical applications, various factors can influence response time and throughput, including bus bandwidth, memory bandwidth, core count, CUDA kernel launch time, and others. The limiting factor can vary significantly depending on the application. To account for this variability, multiple application scenarios are considered. This chapter first mentions how these configurations are implemented. Next the results are evaluated. Finally, a conclusion is drawn which should give more insight in how GPU and GPUDirect could improve application which run on the ONIX system.

5.1. Implementation of Application Scenarios

A single algorithm is employed across different scenarios, with parameters adjusted to simulate varying conditions. The algorithm, derived from Konstantinidis and Cotronis [19], effectively estimates the runtime of different types of GPU kernels. The algorithm can be described as a multiply-add-accumulate function, as illustrated in Listing 5.1. Given its straightforward and highly parallelizable nature, the algorithm allows for a comprehensive comparison of CPU and GPU performance.

Listing 5.1: Multiply-Add-Accumulate Kernel

```
template <ssize_t compute_iterations>
int kernel(float* vec, int vec_len, float seed) {
    int sum = 0;
    for(int i = 0; i < vec_len; i++) {
        for(int j = 0; j < compute_iterations; j++) {
            vec[i] = vec[i] * vec[i] + seed;
        }
        sum += vec[i];
    }
    return sum;
}
```

The evaluation involves four distinct scenarios, determined by varying two parameters: the data size transferred from the FPGA and the data size used in computation. For each of the four scenarios the Arithmetic Intensity (AI) of the computation is also changed. The first parameter, the data size, has a significant impact on PCIe bus throughput. The same data sizes are used as in Chapter 4, small data sizes are set at 4 KB, while large data sizes are set at 10 MB. The second parameter, the computation data size, affects memory bandwidth relative to bus bandwidth. Scenarios with small memory bandwidth use data sizes identical to the transferred data, whereas large memory bandwidth scenarios use data sizes 100 times greater than the transferred data. The memory bandwidth is increased by doing the computation multiple times on the the same data. The final parameter that is changed is the AI, which measures the number of operations performed per byte moved from memory to a core, encompassing operations like additions or multiplications. AI is thus defined in Floating Point Operations (FLOPs) per byte. Typically, AI is low; for example, a floating-point vector addition has an AI of 0.083 FLOPs/byte, while dense linear algebra can achieve AI values in the order of tens, for large matrices. By varying the AI and calculating throughput, a roofline graph can be constructed. This graph helps identify the limiting hardware component, whether it be memory bandwidth, PCIe bus speed or computational throughput of the CPU or GPU. The bandwidth limits for different components are summarized in Table 5.1. Giga Floating Point Operations (GFLOPS) for a given bandwidth can be calculated by multiplying AI with the bandwidth in GB/s. The hardware used in the experiments is the same as in Chapter 4 and is thus again summarized by Table 4.1. The bandwidth limits for the CPU and GPU were determined using the original application by Konstantinidis and Cotronis [19]. The PCIe bus limits are taken from Chapter 4. An overview of these scenarios is presented in Table 5.2.

Table 5.1: Bandwidth Limits of Different Components

Component	**Limit**
CPU (4 cores enabled)	
Compute BW	410 GFLOPS
Memory BW	40 GB/s
GPU	
Compute BW	1300 GFLOPS
Memory BW	75 GB/s
PCIe Bus (4kB)	
Write BW	161 MB/s
Read BW	168 MB/s
PCIe Bus (10MB)	
Write BW	1572 MB/s
Read BW	1705 MB/s

Table 5.2: Summary of Experiments

Experiment ID	**Data Transfer Size**	**Computation Size**	**Sequence of Operations**
1	4 kB	4 kB	Read 4 kB data → Compute on 4 kB data → Send 4 kB back
2	4 kB	400 kB	Read 4 kB data → Compute on 400 kB data → Send 4 kB back
3	10 MB	10 MB	Read 10 MB data → Compute on 10 MB data → Send 10 MB back
4	10 MB	1 GB	Read 10 MB data → Compute on 1 GB data → Send 10 MB back

Two implementations were developed for testing: one in standard C++ and another using the CUDA framework. The C++ implementation runs entirely on the CPU, with data transfer and processing specified as program inputs. It employs three threads: one for receiving data, one for processing, and one for sending data.

The CUDA implementation mirrors this setup but performs computations within a GPU kernel. For the CUDA version, two methods are tested: data transfer through host memory and pre-pinned GPUDirect memory. In the host-memory transfer method, five threads are utilized—two additional threads handle the data transfer to and from the GPU. In contrast, the GPUDirect implementation uses only three threads. Both methods employ a double-buffering technique to maximize throughput, where one buffer is used for computation while the other is available for data transfer. This technique ensures that throughput is limited by the slowest thread, while latency remains unaffected.

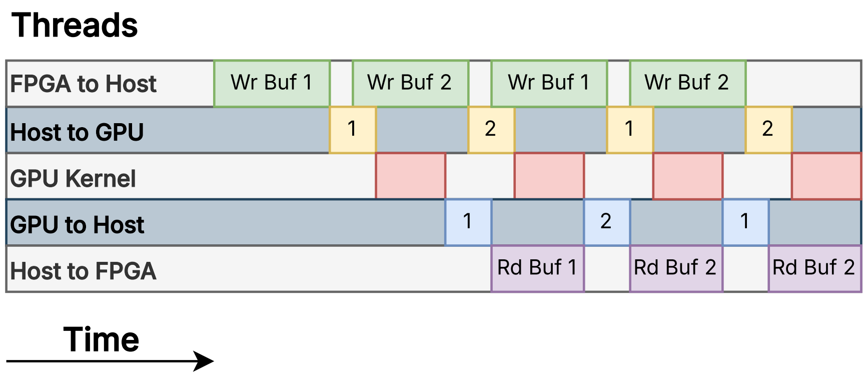


Figure 5.1: Overview of closed loop data pipeline.

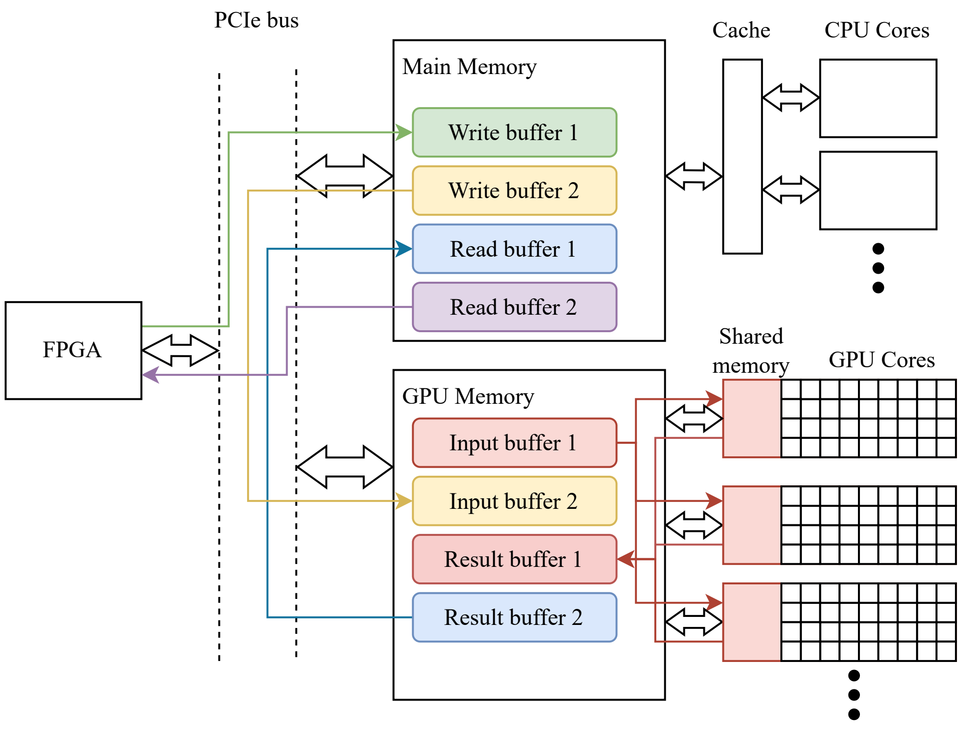


Figure 5.2: Memory locations and their interactions for closed loop data pipeline.

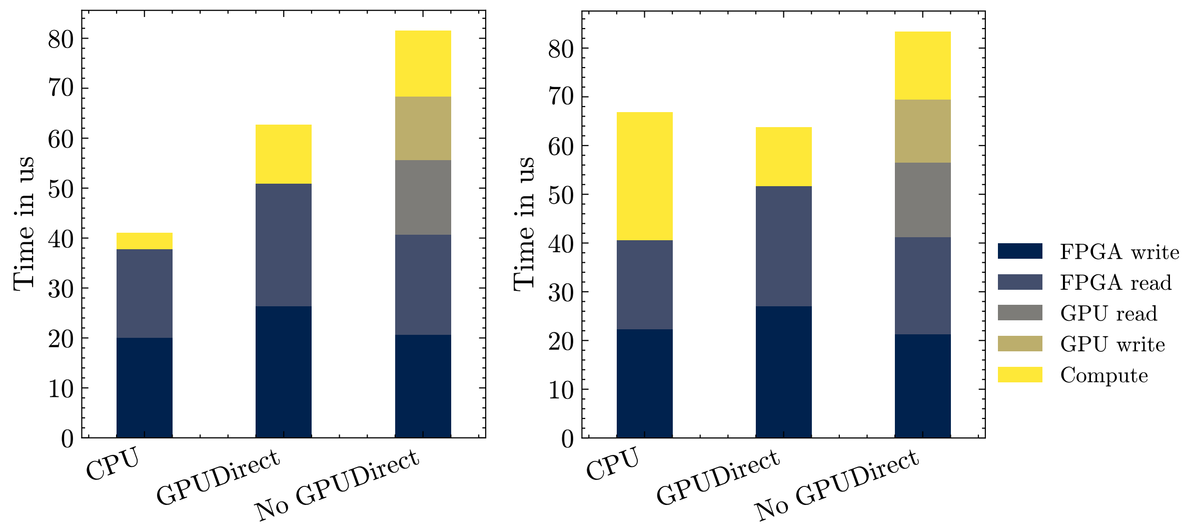
The processing algorithm is optimized for speed by leveraging parallelism. In the C++ implementation, this is achieved trough OpenMP pragmas. The data that is send by the FPGA is interpreted as a one dimensional array. The array is subdivided into blocks of 256 elements, processed by multiple threads using `#pragma omp for`. Within these blocks, a second loop performs eight multiply-accumulate operations per cycle, parallelized using SIMD instructions with `#pragma omp simd`. The CPU utilizes 512-bit AVX registers.

For the CUDA implementation, the array is divided among different blocks, each consisting of 64 cuda threads. These blocks are processed by streaming multiprocessors (SMs), which have multiple cores and memory. Every thread calculates four elements; the number of compute iterations is adjusted to vary the arithmetic intensity (AI).

5.2. Results

5.2.1. Small Data Transfer, Small Computation

The first experiment involves the following sequence of operations: Read 4 kB of data → Compute on 4 kB of data → Send 4 kB back. The plot in Figure 5.3a shows the roundtrip time at an AI of 0.75. The results indicate that the FPGA and GPU read and write times, put together is the primary limiting factor in this scenario, as it dominates the total response time. The additional read and write operations for the non-GPUDirect version significantly slow down the response compared to the GPUDirect version, making the GPUDirect version 25% faster. This suggests that leveraging GPUDirect can substantially improve performance when utilizing the GPU's computational capabilities. However, the CPU is the fastest computation platform in this case, being around 35% faster then using the GPU with GPUDirect. The small amount of data seems to be better handled by the CPU. This is because even though the GPU has a higher maximum bandwith, the CPU is generally faster with computing small data sets as it has a more optimized computer architecture for this. Also, the transfer to the CPU is slightly faster then to the GPU. This shows the the transfer to GPU memory gives a slight performance penalty when compared to sending data just to the main memory. The round-trip time for an AI of 10 is shown in Figure 5.3b. Here, the GPU is 60% faster than the CPU, indicating that once the AI goes up, the GPU is able to slightly outperform the CPU, even at small data sizes. This is because the total amount of parallel computations increased which the GPU is better optimized for.



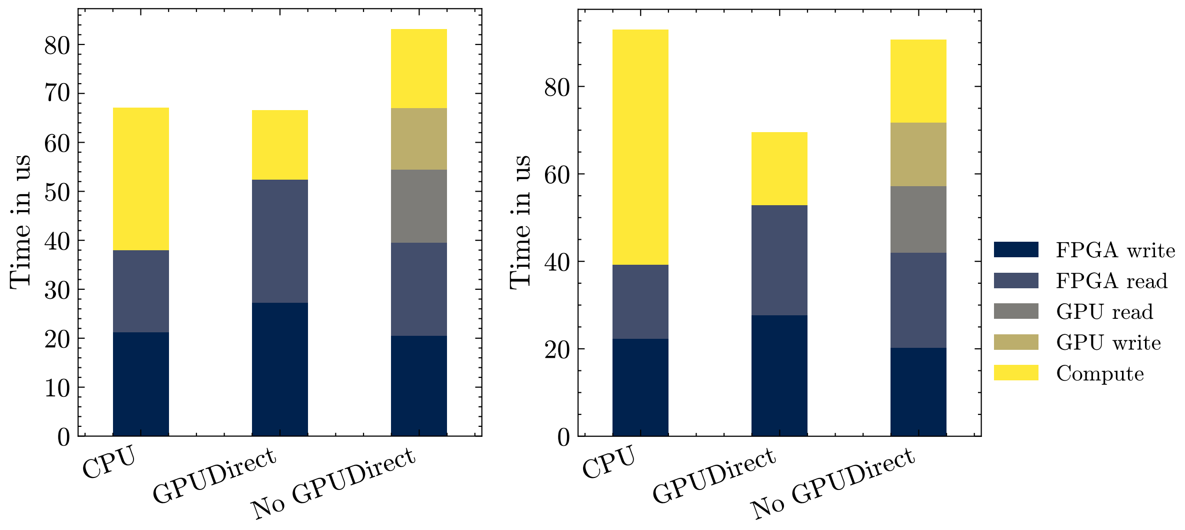
(a) Roundtrip time 4kB Package, 4kB Compute, 0.75 AI

(b) Roundtrip time 4kB Package, 4kB Compute, 10 AI0

Figure 5.3: Round-trip time for 4 kB package with 400 kB of processing

5.2.2. Small Data transfer, Large Computation

In the second experiment, the sequence of operations is: Read 4 kB of data → Compute on 400 kB of data → Send 4 kB back. Figure 5.4a shows the transfer time at AI 0.75 and Figure 5.4b at AI 10. Even tough the total amount that should be computed is 100 larger, the compute time for the GPU is exactly the same as in previous experiment. The CPU is impacted by the additional computational load. For an AI of 0.75 the CPU and GPUDirect version perform similar. When the AI increases to 10, the GPUDirect version is around 25% faster then the CPU version.



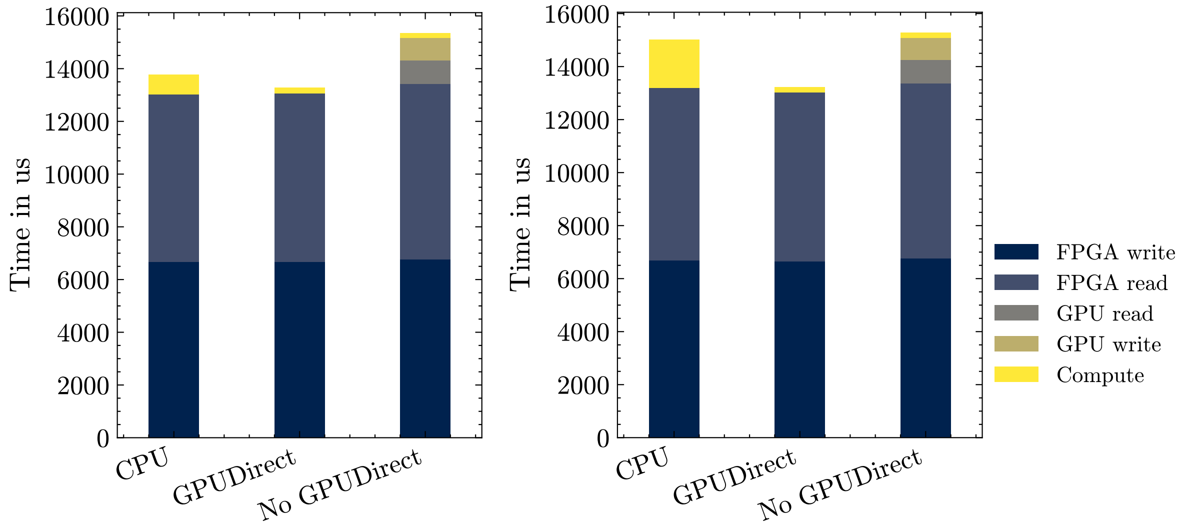
(a) Roundtrip time 4kB Package, 400kB Compute, 0.75 AI

(b) Roundtrip time 4kB Package, 400kB Compute, 10 AI

Figure 5.4: Round-trip time for 4 kB package with 400 kB of processing

5.2.3. Large Data Transfer, Small Computation

The third experiment involves larger data transfers: Read 10 MB of data → Compute on 10 MB of data → Send 10 MB back. Figure 5.5 shows the round trip time at an AI of 0.75 and 10. For these amounts of data, the transfer to and from the FPGA becomes by far the largest contributor to the overhead. Since this is the same for all three configurations, the relative difference between them becomes smaller. The GPUDirect configuration is the fastest, being around 15% faster than no GPUDirect and 3% faster when using the CPU. Increasing the AI does not affect the computation time of the GPU kernel. This is likely because with a small AI the GPU resources are under utilized, meaning that the total compute time mostly comprises of overheads and adding additional computation has no significant effect on the compute time. Increasing the AI does significantly increase the CPU computation time, causing the response time to be 10% slower than GPUDirect.



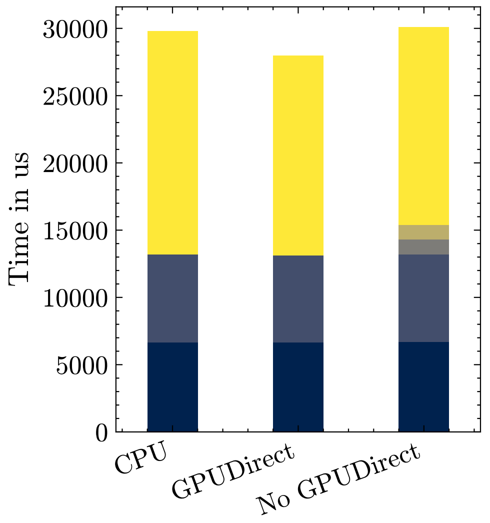
(a) Roundtrip time 10MB Package, 10MB Compute, 0.75 AI

(b) Roundtrip time 10MB Package, 10MB Compute, 10 AI

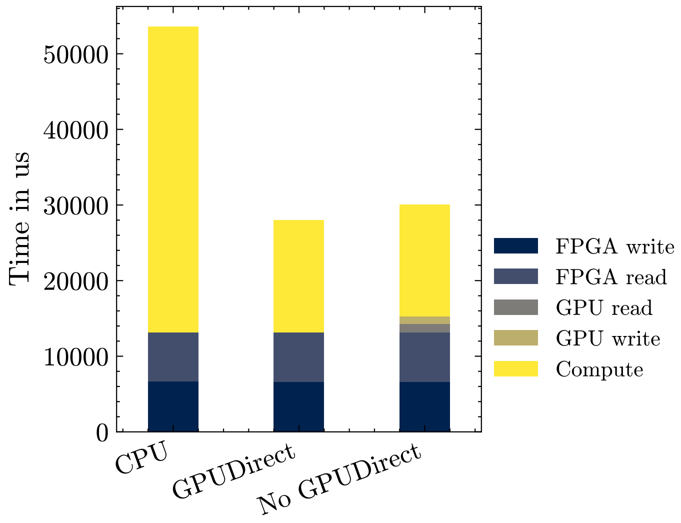
Figure 5.5: Round-trip time for 10 MB package with 10 MB of processing

5.2.4. Large Data Transfer, Large Computation

In the final experiment, each block of 10 MB data is processed with 100 times, totaling 1 GB: Read 10 MB of data → Compute on 1 GB of data → Send 10 MB back. The total round trip time for the two tested AIs is shown in Figure 5.6. At 1GB of data compute, the compute time becomes the dominant factor in the total response time. For both cases, the GPU is the better compute platform. However, when comparing both GPU configurations the difference is less pronounced due to the fact that the total contributions of the additional read and writes become smaller. In both cases, the GPUDirect configuration is only 6% faster than not using GPUDirect.



(a) Roundtrip time 10MB Package, 1GB Compute, 0.75 AI



(b) Roundtrip time 10MB Package, 1GB Compute, 10 AI

Figure 5.6: Round-trip time for 10 MB package with 1 GB of processing

5.3. Conclusion

The experiments conducted in this chapter systematically investigated the performance implications of various configurations and parameters on total round trip time in a closed-loop system. The primary objective was to determine in which application scenarios a GPU and GPUDirect offers improved performance. While it is challenging to provide a definitive answer, the results offer a general understanding of how data size and throughput impact different aspects of of the total closed loop cycle.
The findings indicate that for the smallest transfer size of 4 KB, the GPU provides minimal performance benefits compared to CPU computation. Although GPUDirect significantly accelerates the data transfer from FPGA to GPU, its utility is questionable in this context. In this specific algorithm, which is simple and could be easily accelerated by a GPU, there is no noticeable performance improvement due to the small data sizes. Even with an increase in algorithm complexity, the performance gain remains marginal.
For larger data transfers, GPU's can have a improved performance compared to the CPU. However, for GPUDirect the pattern observed in Chapter 4 re-emerges: the relative performance improvement diminishes as transfer sizes increase. This is attributable to the limited throughput capability of the FPGA's PCIe interface. When 10 MB of data is used in computation, the GPU's impact is minimal, as the computation time is negligible compared to the data transfer time.
Thus, the benefits of incorporating GPUDirect into the system should be carefully evaluated. Given the increased complexity and minimal performance gains observed in this study, it may be worth reconsidering whether the effort to integrate GPUDirect justifies the added complexity and potential performance improvements.

6 Discussion and Future Directions

The thesis thus far has analyzed the current design of the ONIX system and proposed potential improvements, primarily focusing on software enhancements and changes on the user desktop. The first part of this chapter consolidates these findings, explores their use cases, and evaluates their practical applications. The second part looks into hardware changes of the ONIX FPGA card. While the improvements discussed so far are software-oriented, achieving optimal performance may require hardware-level modifications to the ONIX system—especially when integrating additional high-speed interfaces. Therefore, this chapter also investigates potential hardware upgrades that could further enhance the system's capabilities. Finally, the chapter provides a comparative analysis of the GPUDirect implementation presented in this thesis with other implementations, along with a brief discussion on its applicability to other domains.

6.1. Cases for Utilizing GPUDirect

The decision tree depicted in Figure 6.1 outlines the considerations for determining whether GPUDirect can provide significant speedup for a given application. Users should begin by evaluating the parallelizability of their application. If the application is not highly parallelizable, GPUs offer no advantage, and efforts should focus on optimizing CPU performance using SIMD registers and multithreading.

Next, the decision tree considers whether the data and algorithm size are large (greater than approximately 1MB). If the size is not large, the focus should again be on CPU optimization techniques. However, if the size is large, the decision tree then examines whether the data transfer constitutes a significant portion of the total processing time. If the transfer is not a large part, efforts should be directed towards algorithm speedup.

The user should also evaluate whether the GPU in their system PCIe capability is significantly better than that of ONIX. If it is, GPUDirect might provide a slight speedup. Conversely, if the GPU's PCIe capability is not much better than ONIX, GPUDirect could offer a more substantial speedup. But this is the biggest problem for the GPUDirect solution. Almost all GPU's have much better PCIe capabilities than the ONIX card. Even the NVIDIA Quadro P620 used in this study, which is a relatively old and lightweight card had a much faster PCIe connection. This implies that for GPUDirect to make sense in future systems the hardware capability of the ONIX FPGA card should be improved first.

The decision tree helps identify the scenarios where GPUDirect would provide the most benefit and where alternative approaches would be more effective. This approach ensures that resources and efforts are directed towards achieving the highest performance improvements possible under the given constraints.

6.2. Potential Upgrades to ONIX System Components

If the ONIX would support higher throughput ephys devices parts of the system should be upgraded to handle the higher data loads. When assessing which parts to update it is important that the most limiting components of the hardware are revised first to this end a overview of which parts could be limiting the data throughput on the ONIX FPGA Card is shown in Figure 6.2. The ONIX system's

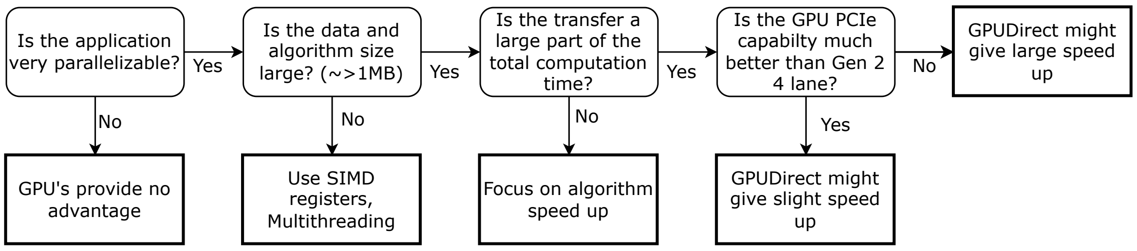

Is the application very parallelizable?
Yes
Is the data and algorithm size large? (~>1MB)
Yes
Is the transfer a large part of the total computation time?
Yes
Is the GPU PCIe capabilty much better than Gen 2 4 lane?
No
GPUDirect might give large speed up
No
GPU's provide no advantage
No
Use SIMD registers, Multithreading
No
Focus on algorithm speed up
Yes
GPUDirect might give slight speed up

Figure 6.1: Decision Tree for Evaluating GPUDirect Speedup Potential

performance is currently limited by the Serializer/Deserializer (SerDes) modules on the FMC card, which interface with the electrophysiology devices and the Nereid PCIe board. The system employs the DS90UB933/DS90UB934 SerDes pair [12, 13], with two deserializers supporting up to 300 MB/s of combined bandwidth. Upgrading to higher-speed SerDes modules such as the DS90UB934-Q1 (up to 13.5 Gbps) or TSER9615 (up to 7.55 Gbps) [38] could significantly enhance data throughput, although this would require addressing challenges related to signal integrity at higher speeds.

Another approach to improve bandwidth could be to add additional deserializers to the FMC card. The ONIX FMC card connects to the Nereid board via a VITA 57.1 FMC HPC connector with 174 I/O pins, 114 of which are currently utilized. The SerDes modules occupy 25 pins, allowing the potential addition of two more modules within the existing setup. To accommodate further expansion, the Nereid board could be replaced with an FPGA card that supports more I/O pins, such as those with FMC+ connectors compliant with the VITA 57.4 standard. An example is the VCK190 (Versal AI)[41], which features two FMC+ connectors and higher bandwidth capabilities. These cards are particularly suited for high-throughput applications.

To further enhance bandwidth, multi-gigabit transceivers (MGTs) integrated into the FPGA could be leveraged. The Nereid board's FPGA includes four GT lanes supporting data rates of up to 6.6 Gbps. Boards like the Versal AI Core VCK190, which features 12 GT lanes per connector, offer significant upgrades in bandwidth capacity and processing power. Such advancements would enable future iterations of ONIX to better handle next-generation neural interfaces.

Replacing the Nereid PCIe card offers another straightforward upgrade path. The replacement card must meet the following criteria: compliance with the VITA 57.1 or 57.4 standard, inclusion of an FPGA, onboard DRAM, and a PCIe edge connector. An example is the Kintex UltraScale KCU105 [1], which provides an FMC HPC connector and Gen3 x8 PCIe support. This card could double the number of SerDes interfaces on the FMC card. Additionally, cards with advanced PCIe capabilities, such as the Versal AI VEK280 [39] (Gen4 x16 PCIe) [40] or the Versal HBM VHK158 [41] (Gen5 x16 PCIe), could be considered. These upgrades would become critical once the PCIe bus bandwidth becomes a limiting factor, ensuring the ONIX system remains scalable and performant.

While the current FPGA on the Nereid card is not fully utilized (as shown in Table 6.2), upgrading to a higher-performance FPGA would prepare the system for scenarios requiring additional computational resources. For example, increasing the width of datapaths or supporting additional interfaces could demand more FPGA resources. An advanced card like the Virtex UltraScale+ VCU118 [2], offering enhanced logic capacity and bandwidth, could address these requirements effectively.

In summary, incremental upgrades to SerDes modules, leveraging multi-gigabit transceivers, or replacing the Nereid card with a more capable FPGA card are all viable pathways to improving the ONIX system. These upgrades would enable higher data throughput, better scalability, and preparation for future neural interface applications.

6.3. Comparison of GPUDirect Solution to Previous Work

Although the proposed communication solution in this thesis does not demonstrate a significant performance improvement for the ONIX system, it does not imply that it cannot be useful in other contexts. In Section 2.11, several previous implementations of FPGA-to-GPU PCIe P2P communication were discussed. A comparison of the maximum achieved throughput in each case reveals significant differences in performance. Given the varying number of PCIe lanes and PCIe generations used in each solution, a more meaningful comparison can be made by examining the utilization of the PCIe lanes.

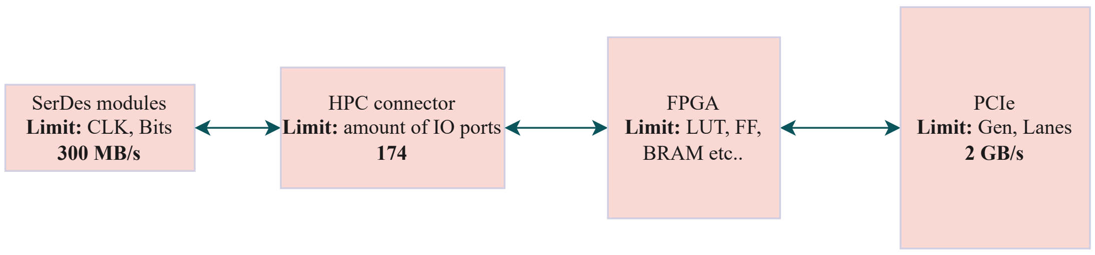


Figure 6.2: Overview of limiting hardware specs on the ONIX PCIe host

Resource	Utilization	Available	Utilization (%)
LUT	41,776	101,400	41.20
LUTRAM	2,192	35,000	6.26
FF	52,979	202,800	26.12
BRAM	123.50	325	38.00
DSP	2	600	0.33
IO	235	400	58.75

Table 6.1: FPGA Resource Utilization

The FPGA^2 solution performs well in this regard, achieving a utilization of 81% and 75% for FPGA-to-GPU and GPU-to-FPGA communication, respectively.[36] However, this solution does not scale effectively as it is implemented for only one lane. The solution proposed by Bittner & Ruf shows decent performance, with an 80% utilization in the GPU-to-FPGA direction, but significantly lower performance in the FPGA-to-GPU direction.[4] Notably, it is the only solution that operates under Windows, making it the only viable option if that is a constraint.

Kasai & Osana's solution achieves the highest utilization in the FPGA-to-GPU direction, with 81%.[16] However, it does not successfully implement GPU-to-FPGA communication. This limitation could be attributed to the fact that the PCIe bridges on their motherboard did not support PCIe P2P read transactions. Through the work conducted in this thesis, it was observed that PCIe write transactions are often supported, as they are simply forwarded to a specified memory address by PCIe bridges, making P2P writes functional. However, for P2P reads, the read response needs to be routed back to the source by the PCIe bridges, a feature often not supported by many motherboards.

The work presented in this thesis demonstrates that an utilization of 79% and 86% can be achieved using a modified RIFFA driver, representing the best utilization in the GPU-to-FPGA direction among the compared works, and a comparable result in the FPGA-to-GPU direction. Considering that RIFFA is

Board	Device Name	FMC Type	PCIe Version	PCIe Lanes
VCK190	Versal AI	2 FMC+	Gen4	8
Kintex UltraScale KCU105	Kintex UltraScale	HPC	Gen3	8
Versal Prime VMK180	Versal Prime	2 FMC+	Gen4	8
Virtex UltraScale+ VCU118	Virtex UltraScale+	HPC	Gen4	8
Versal AI VEK280	Versal AI	FMC+	Gen4	16
Versal HBM VHK158	Versal HBM	FMC+	Gen5	16

Table 6.2: Specifications of FPGA Development Boards Mentioned for ONIX System Upgrades

an open-source communication library and easy to interface with, the proposed solution in this thesis—combining GPUDirect with RIFFA—proves to be a promising approach for developers seeking lower latency on the PCIe bus.

	Previous Work			This Thesis
	Bittner & Ruf [4]	FPGA^2 [36]	Kasai & Osana [16]	
Operating System	Windows	Linux	Linux	Linux
DMA Master	GPU	FPGA	FPGA	FPGA
FPGA Vendor	Xilinx	Xilinx	Xilinx	Xilinx
FPGA Model	Virtex 6	Virtex 5	Alveo U50	Kintex-7
FPGA IP Stack	Custom	Custom	Vendor	RIFFA
FPGA Driver	Custom	Custom	XDMA	Modified RIFFA
FPGA Programming	HDL	HDL	HDL	HDL
GPU Vendor	NVIDIA	NVIDIA	NVIDIA	NVIDIA
GPU Model	GeForce GTX580	GeForce 8400GS	Quadro RTX 4000	Quadro P600
GPU Driver	Original	Nouveau	Original	Original
GPU Programming	CUDA	gdev	CUDA	CUDA
Effective PCIe Lanes	8	1	4	4
PCIe Generation	1.0	1.0	3.0	2.0
Maximal Throughput (FPGA to GPU)	514 MB/s	203 MB/s	32500 MB/s	1583 MB/s
Maximal Throughput (GPU to FPGA)	1.6 GB/s	189 MB/s	N/A	1728 MB/s

Table 6.3: Comparison of Previous Work with This Thesis

6.4. Conclusion

This chapter has explored the potential enhancements to the ONIX system, focusing on both software and hardware improvements, and evaluated their practical implications for next-generation neural interfaces. By analyzing the scenarios in which GPUDirect offers significant performance gains, a structured decision-making framework was presented to guide users in determining its applicability. The findings highlight that while GPUDirect provides benefits under certain conditions, but with the current hardware it would not be logical to use. The ONIX system's current limitations are primarily in PCIe bandwidth and SerDes.

Hardware upgrades were identified as critical to addressing these bottlenecks. Incremental improvements to the FMC card, such as integrating higher-speed SerDes modules or additional deserializers, present viable pathways to enhance bandwidth. Replacing the Nereid card with advanced FPGA boards or leveraging multi-gigabit transceivers could further boost data throughput and ensure scalability for future neural interface applications. The proposed upgrades emphasize a modular approach, enabling the ONIX system to adapt to evolving experimental demands.

Finally, the comparison of GPUDirect implementations contextualizes the work presented in this thesis within the broader research landscape. By achieving competitive PCIe lane utilization using an open-source driver, the solution demonstrates promise for applications requiring efficient GPU-FPGA communication. However, the analysis underscores the importance of aligning system design with application-specific requirements, especially when integrating advanced computational methods.

In summary, this chapter has provided a roadmap for long-term improvements to the ONIX system. By addressing current hardware constraints and refining software capabilities, the ONIX platform can better support high-throughput neural interfaces and computational algorithms, paving the way for advancements in electrophysiology research.

7 Conclusion

7.1. Summary

This thesis investigated the feasibility and potential benefits of integrating GPU computation with the Open Ephys ONIX system to enhance its capacity for next-generation neural interfaces and computational algorithms. The research addressed the overarching question:

How can GPU computation be effectively integrated with the ONIX electrophysiology system to enhance support for next-generation neural interfaces and computational algorithms?

To answer this question, the study pursued several specific objectives. The findings are summarized as follows:

In **Chapter 2**, we provided the foundational background on the ONIX system, detailing its architecture, hardware components, and libraries. This ensured that the GPU integration respected the system's modular and flexible framework. Additionally, data transfer methods and kernel execution techniques were reviewed to balance latency, throughput, and usability. A survey of related literature contextualized the contributions of this thesis within the broader field of neural data acquisition and processing.

Chapter 3 focused on implementing key modifications for GPU integration. This included the development of a secondary data channel on the FPGA and a custom kernel driver that combined GPUDirect functionality with the RIFFA driver. Updates to the Liboni API were made to support these changes.

In **Chapter 4** Benchmarks are done. First, the modifications to the FPGA logic are tested without GPUDirect and it shows improvement compared to the original system. Next unpinned and pinned GPUDirect transfer strategies are compared to the improved system which showed that pinned transfers improved throughput by 30% for small transfers and 14% for larger ones. However, significant jitter was observed across all configurations, emphasizing the challenge of achieving consistent performance.

In **Chapter 5**, we analyzed the performance of GPUDirect in closed-loop applications. The evaluation showed that GPU acceleration provided benefits for larger transfers and higher arithmetic intensity. However, limited PCIe bandwidth in the ONIX system constrained the benefits of using GPUDirect, with CPU-based implementations outperforming GPUs for smaller transfers. These results demonstrated that GPU acceleration is most effective when applied selectively, depending on data size and computational intensity.

Chapter 6 addressed the hardware limitations of the ONIX system, including PCIe bandwidth constraints, FPGA resources, and Serializer/Deserializer (SerDes) module capabilities. Proposed hardware upgrades, such as higher-speed SerDes modules and advanced FPGA cards with improved PCIe support, were outlined to address these bottlenecks. A decision tree framework was developed to help users evaluate when GPUDirect integration offers meaningful performance improvements. Comparisons with prior GPUDirect implementations highlighted this work's competitive lane utilization and broader applicability beyond the ONIX system.

Bringing these findings together, the thesis concludes that GPU integration demonstrates potential benefits in specific scenarios but is currently constrained by the hardware limitations of the ONIX system. Achieving the goal of supporting next-generation neural interfaces requires a multifaceted approach that includes GPU integration, hardware improvements, and algorithmic optimization.

7.2. Main Contributions

The main contributions of this thesis are as follows:

- **Comprehensive Analysis of the ONIX System:** Detailed evaluation of the ONIX architecture, identifying critical hardware and software limitations.
- **Design of GPU Integration:** Development of a secondary data channel on the FPGA, a custom kernel driver with GPUDirect and RIFFA integration, and updates to the Liboni API for seamless system compatibility.
- **Performance Evaluation:** Benchmarking of pinned and unpinned transfer strategies and analysis of GPUDirect performance in closed-loop applications, revealing strengths and constraints.
- **Hardware Recommendations:** Identification of ONIX system bottlenecks and proposals for incremental hardware upgrades to enhance throughput and scalability.
- **Decision Tree Framework:** Creation of a tool to guide researchers in determining the conditions under which GPUDirect provides meaningful benefits.
- **Broader Applicability of GPUDirect:** Demonstration of the GPUDirect implementation's competitive lane utilization and its potential for use beyond the ONIX system.

7.3. Future Work

Building upon the contributions of this thesis, several directions for future research are proposed:

- **Hardware Upgrades:** Implement higher-bandwidth Serializer/Deserializer (SerDes) modules, advanced FPGA cards with PCIe Gen4 or Gen5 capabilities, and additional deserializers to address throughput limitations.
- **Reducing System Jitter:** Investigate techniques to minimize jitter and improve the reliability of data transfer for real-time neuroscience applications.
- **Tailored Computational Solutions:** Conduct a detailed analysis of the computational requirements of ephys workflows to guide the design of optimized hardware and software solutions.
- **Expanded Applications of GPUDirect:** Explore the use of GPUDirect in other high-bandwidth, low-latency domains, such as position tracking or real-time data processing tasks.
- **Scalability Studies:** Evaluate the scalability of the system on GPUs with higher computational capacity or alternative acceleration platforms to further enhance performance.

By addressing these directions, future work can overcome the current limitations and push the boundaries of what is possible with electrophysiology systems. This progress will support more advanced neuroscience research and the development of next-generation neural interfaces.

Bibliography

[1] *AMD Kintex UltraScale FPGA KCU105 Evaluation Kit*. en. URL: `https://www.xilinx.com/products/boards-and-kits/kcu105.html` (visited on 11/19/2024).

[2] *AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit*. en. URL: `https://www.xilinx.com/products/boards-and-kits/vcu118.html` (visited on 11/19/2024).

[3] . *An FPGA IP core for easy DMA over PCIe with Windows and Linux | xillybus.com*. URL: `https://xillybus.com/` (visited on 08/08/2024).

[4] Ray Bittner and Erik Ruf. "Direct GPU/FPGA Communication Via PCI Express". en. In: ().

[5] Alessio P. Buccino, Samuel Garcia, and Pierre Yger. "Spike sorting: new trends and challenges of the era of high-density probes". en. In: *Progress in Biomedical Engineering* 4.2 (May 2022). Publisher: IOP Publishing, p. 022005. ISSN: 2516-1091. DOI: `10.1088/2516-1091/ac6b96`. URL: `https://dx.doi.org/10.1088/2516-1091/ac6b96` (visited on 08/06/2024).

[6] Denise J. Cai et al. "A shared neural ensemble links distinct contextual memories encoded close in time". en. In: *Nature* 534.7605 (June 2016). Publisher: Nature Publishing Group, pp. 115–118. ISSN: 1476-4687. DOI: `10.1038/nature17955`. URL: `https://www.nature.com/articles/nature17955` (visited on 08/07/2024).

[7] Zhe Sage Chen and Bijan Pesaran. "Improving scalability in systems neuroscience". English. In: *Neuron* 109.11 (June 2021). Publisher: Elsevier, pp. 1776–1790. ISSN: 0896-6273. DOI: `10.1016/j.neuron.2021.03.025`. URL: `https://www.cell.com/neuron/abstract/S0896-6273(21)00195-1` (visited on 08/23/2024).

[8] Davide Ciliberti and Fabian Kloosterman. "Falcon: a highly flexible open-source software for closed-loop neuroscience". en. In: *Journal of Neural Engineering* 14.4 (June 2017). Publisher: IOP Publishing, p. 045004. ISSN: 1741-2552. DOI: `10.1088/1741-2552/aa7526`. URL: `https://dx.doi.org/10.1088/1741-2552/aa7526` (visited on 08/09/2024).

[9] *Getting Started — ONIX Docs*. URL: `https://open-ephys.github.io/onix-docs/Getting%20Started/index.html` (visited on 08/23/2024).

[10] *Home — ONIX Docs*. URL: `https://open-ephys.github.io/onix-docs/index.html` (visited on 12/05/2023).

[11] Guosong Hong and Charles M. Lieber. "Novel electrode technologies for neural recordings". en. In: *Nature Reviews Neuroscience* 20.6 (June 2019). Publisher: Nature Publishing Group, pp. 330–345. ISSN: 1471-0048. DOI: `10.1038/s41583-019-0140-6`. URL: `https://www.nature.com/articles/s41583-019-0140-6` (visited on 08/23/2024).

[12] Texas Instruments. *DS90UB933-Q1 12-Bit, 100-MHz FPD-Link III Deserializer for 1MP/60fps and 2MP/30fps Cameras*. URL: `https://www.ti.com/lit/gpn/ds90ub933-q1` (visited on 08/07/2024).

[13] Texas Instruments. *DS90UB934-Q1 12-Bit, 100-MHz FPD-Link III Deserializer*. URL: `https://www.ti.com/lit/gpn/ds90ub934-q1` (visited on 08/07/2024).

[14] LLC Intan Technologies. *RHD2000 Series Digital Electrophysiology Interface Chips*. Nov. 1012. URL: `https://intantech.com/files/Intan_RHD2000_series_datasheet.pdf` (visited on 08/08/2024).

[15] Shun Kasai and Yasunori Osana. "A driver-based approach for DMA transfer between FPGA-GPU". In: *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*. ISSN: 2832-1324. Nov. 2022, pp. 103–108. DOI: `10.1109/CANDARW57323.2022.00061`.

[16] Shun Kasai and Yasunori Osana. “A driver-based approach for DMA transfer between FPGA-GPU”. In: *2022 Tenth International Symposium on Computing and Networking Workshops (CANDARW)*. ISSN: 2832-1324. Nov. 2022, pp. 103–108. DOI: `10.1109/CANDARW57323.2022.00061`.

[17] S. Kato. “Implementing Open-Source CUDA Runtime”. In: 2013. URL: `https://www.semanticscholar.org/paper/Implementing-Open-Source-CUDA-Runtime-Kato/13efdbff6365e5ff6337ad3b672` (visited on 10/21/2024).

[18] Justin P. Kinney et al. “A direct-to-drive neural data acquisition system”. English. In: *Frontiers in Neural Circuits* 9 (Sept. 2015). Publisher: Frontiers. ISSN: 1662-5110. DOI: `10.3389/fncir.2015.00046`. URL: `https://www.frontiersin.org/journals/neural-circuits/articles/10.3389/fncir.2015.00046/full` (visited on 08/09/2024).

[19] Elias Konstantinidis and Yiannis Cotronis. “A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling”. In: *Journal of Parallel and Distributed Computing* 107 (Sept. 2017), pp. 37–56. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2017.04.002`. URL: `https://www.sciencedirect.com/science/article/pii/S0743731517301247` (visited on 07/03/2024).

[20] Gonçalo Lopes et al. “Bonsai: an event-based framework for processing and controlling data streams”. English. In: *Frontiers in Neuroinformatics* 9 (Apr. 2015). Publisher: Frontiers. ISSN: 1662-5196. DOI: `10.3389/fninf.2015.00007`. URL: `https://www.frontiersin.org/journals/neuroinformatics/articles/10.3389/fninf.2015.00007/full` (visited on 08/06/2024).

[21] Rufus Mitchell-Heggs et al. “Neural manifold analysis of brain circuit dynamics in health and disease”. en. In: *Journal of Computational Neuroscience* 51.1 (Feb. 2023), pp. 1–21. ISSN: 1573-6873. DOI: `10.1007/s10827-022-00839-3`. URL: `https://doi.org/10.1007/s10827-022-00839-3` (visited on 05/03/2023).

[22] Carolina Mora Lopez et al. “A Neural Probe With Up to 966 Electrodes and Up to 384 Configurable Channels in 0.13 μm SOI CMOS”. In: *IEEE Transactions on Biomedical Circuits and Systems* 11.3 (June 2017). Conference Name: IEEE Transactions on Biomedical Circuits and Systems, pp. 510–522. ISSN: 1940-9990. DOI: `10.1109/TBCAS.2016.2646901`. URL: `https://ieeexplore.ieee.org/document/7900417` (visited on 08/07/2024).

[23] Narendra Mukherjee, Joseph Wachutka, and Donald Katz. *Python meets systems neuroscience: affordable, scalable and open-source electrophysiology in awake, behaving rodents*. Pages: 105. Jan. 2017. DOI: `10.25080/shinma-7f4c6e7-00e`.

[24] Jonathan P. Newman et al. *A unified open-source platform for multimodal neural recording and perturbation during naturalistic behavior*. en. Pages: 2023.08.30.554672 Section: New Results. Sept. 2023. DOI: `10.1101/2023.08.30.554672`. URL: `https://www.biorxiv.org/content/10.1101/2023.08.30.554672v1` (visited on 03/22/2024).

[25] *nouveau · freedesktop.org*. en. Aug. 2024. URL: `https://nouveau.freedesktop.org/` (visited on 10/21/2024).

[26] NVIDIA. *GPUDirect RDMA*. Aug. 2024. URL: `https://docs.nvidia.com/cuda/gpudirect-rdma/` (visited on 08/24/2024).

[27] Yogi A. Patel et al. “Hard real-time closed-loop electrophysiology with the Real-Time eXperiment Interface (RTXI)”. en. In: *PLOS Computational Biology* 13.5 (May 2017). Publisher: Public Library of Science, e1005430. ISSN: 1553-7358. DOI: `10.1371/journal.pcbi.1005430`. URL: `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1005430` (visited on 08/09/2024).

[28] PCI-SIG. *PCI Express Base Specification Revision 2.0*. Dec. 2006. URL: `https://members.pcisig.com/wg/PCI-SIG/document/download/8246`.

[29] *riffa*. original-date: 2015-05-04T21:24:17Z. July 2023. URL: `https://github.com/KastnerRG/riffa` (visited on 07/12/2023).

[30] Marek Rupniewski et al. *A Real-Time Embedded Heterogeneous GPU/FPGA Parallel System for Radar Signal Processing*. Pages: 1197. July 2016. DOI: `10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0182`.

[31] Bosch Sensortec. *BNO055 Intelligent 9-axis absolute orientation sensor*. Oct. 2021. URL: `https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf` (visited on 08/07/2024).

[32] Joshua H. Siegle et al. “Open Ephys: an open-source, plugin-based platform for multichannel electrophysiology”. en. In: *Journal of Neural Engineering* 14.4 (June 2017). Publisher: IOP Publishing, p. 045003. ISSN: 1741-2552. DOI: `10.1088/1741-2552/aa5eea`. URL: `https://dx.doi.org/10.1088/1741-2552/aa5eea` (visited on 08/09/2024).

[33] Nicholas A. Steinmetz et al. “Neuropixels 2.0: A miniaturized high-density probe for stable, long-term brain recordings”. In: *Science* 372.6539 (Apr. 2021). Publisher: American Association for the Advancement of Science, eabf4588. DOI: `10.1126/science.abf4588`. URL: `https://www.science.org/doi/full/10.1126/science.abf4588` (visited on 08/07/2024).

[34] Ian Stevenson. *Tracking Advances in Neural Recording | Statistical Neuroscience Lab*. en-US. Oct. 2013. URL: `https://stevenson.lab.uconn.edu/scaling/` (visited on 04/29/2024).

[35] Manolis Surligas, Antonis Makrogiannakis, and Stefanos Papadakis. “Maximizing GPU Exploitation for SDR with GPUDirect”. In: *Proceedings of the 2015 Workshop on Software Radio Implementation Forum*. SRIF ’15. New York, NY, USA: Association for Computing Machinery, Sept. 2015, pp. 31–36. ISBN: 978-1-4503-3532-4. DOI: `10.1145/2801676.2801688`. URL: `https://dl.acm.org/doi/10.1145/2801676.2801688` (visited on 08/09/2024).

[36] Yann Thoma, Alberto Dassatti, and Daniel Molla. “FPGA2: An open source framework for FPGA-GPU PCIe communication”. In: *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. ISSN: 2325-6532. Dec. 2013, pp. 1–6. DOI: `10.1109/ReConFig.2013.6732296`.

[37] Uros Topalovic et al. “A wearable platform for closed-loop stimulation and recording of single-neuron and local field potential activity in freely moving humans”. en. In: *Nature Neuroscience* 26.3 (Mar. 2023). Number: 3 Publisher: Nature Publishing Group, pp. 517–527. ISSN: 1546-1726. DOI: `10.1038/s41593-023-01260-4`. URL: `https://www.nature.com/articles/s41593-023-01260-4` (visited on 03/07/2023).

[38] *TSER9615 data sheet, product information and support | TI.com*. URL: `https://www.ti.com/product/TSER9615` (visited on 11/20/2024).

[39] *Versal AI Edge Series VEK280 Evaluation Kit*. URL: `https://www.xilinx.com/products/boards-and-kits/vek280.html` (visited on 11/20/2024).

[40] *Versal HBM Series VHK158 Evaluation Kit*. en. URL: `https://www.xilinx.com/products/boards-and-kits/vhk158.html` (visited on 11/19/2024).

[41] *Versal Prime Series VMK180 Evaluation Kit*. en. URL: `https://www.xilinx.com/products/boards-and-kits/vmk180.html` (visited on 11/19/2024).