Accelerating DSP functions for GNU Radio by implementing them on an FPGA-based PCIe/Thunderbolt co-processor

Author: Victor Omoniyi
Student ID: C21032396
Supervisors: Dr. Frank Langbein & Derek Kozel
MSc Computing

Accelerating DSP functions for GNU Radio by implementing them on an FPGA-based PCIe/Thunderbolt co-processor

ABSTRACT:

GNU Radio is a versatile open-source Python / C++ based software development toolkit used by hobbyists and professionals, which allows the users to interconnect blocks/cores to form a flexible, fully customisable wireless communication system. LiteX is a python-based toolset which allows CPU's or SoCs (System-on-a-chip) to be designed and deployed on a small Field Programmable Gate-Array (FPGA) board. By connecting GNU Radio with LiteX, operations on the host side can be accelerated allowing for faster processing times.

Accelerating DSP functions for GNU Radio by implementing them on an FPGA-based PCIe/Thunderbolt co-processor

ACKNOWLEDGEMENTS:

I would like to take this opportunity to thank my supervisors, Frank Leigbein and Derek Kozel whose expertise and guidance helped shape this project. Without them, I could not have gotten this far. The friendly users on the GNU Radio and LiteX chats who answered my queries in a kind and timely fashion. My family who convinced me to take a leap of faith and to undertake this project and course.

Finally, I'd like to thank Abas, Ellie, and Jess for being a constant source of motivation and for lending their time and ears when times were tough.

Accelerating DSP functions for GNU Radio by implementing them on an FPGA-based PCIe/Thunderbolt co-processor

Table of Contents

LIST OF ABBRIEVATIONS:

OOT: Out of Tree module
SDR: Software-Defined Radio
HDR: Hardware-Defined Radio
FPGA: Field Programmable Gate Array
ASIC: Application-specific integrated circuit
CPU: Central Processing Unit
LUTS: Lookup Tables
PLB: Programmable Logic Blocks
CLB: Configurable Logic Blocks
DFF: D Flip-Flop
SoC: System-on-a-chip
PCIe: Peripheral Component Interconnect Express

Accelerating DSP functions for GNU Radio by implementing them on an FPGA-based PCIe/Thunderbolt co-processor

1 - INTRODUCTION:

In modern day, wireless communication networks have become more and more popular over the years. Advancements in technology since the cellular network have allowed and called for greater flexibility in hardware. Designing and creating new radios can not only be time consuming but can also cost a great amount. Software-Defined Radio's (SDR) have gained popularity due to the increased amount of flexibility compared to traditional Hardware-Defined Radio's (HDR). Furthermore, when coupled with technology such as Field Programmable Gate Arrays (FPGA), the ability to get fast and lightweight SDRs to implement radio function is very achievable for amateurs to professionals.

2 – AIMS & OBJECTIVES:

The aim of this dissertation is to use an FPGA board to connect with GNU Radio and LiteX. We aim for the board to run useful process that will allow for data transfer to be accelerated on GNU Radio. For the speeding up of operations, there are two ways to achieve this. Firstly, by creating and adding blocks that will transport data to and from the card. Secondly, by adding existing blocks such as low pass filter to run on the board using LiteX. The host code will use NumPy to handle the data transport. Ideally, the code will be simplistic and easily customisable. When creating the blocks, it can be as simple or as complicated as the user wants without the code becoming obfuscated. For writing these blocks, I will be using either an out of tree (OOT) modules or an embedded block. OOT modules are simply components that don't live within the GNU Radio source tree. This will allow me to extend GNU Radio with my own functions and blocks, allowing for me to maintain the code.

The dissertation will also aim to explain what other SDR alternatives are currently on the market and the pros and cons of these applications. It will also explain why I have chosen to use GNU Radio and LiteX instead of these other alternatives.

Furthermore, a secondary aim is to analyse the tools used, what is currently on the market and in development and to see whether the processes I am using can be better streamlined and improved in the future with the rise of new software.

3- BACKGROUND:

A software-defined radio (SDR) is a radio communication system where components that may have traditionally been implemented in a hardware, such as an amplifier or a filter, are implemented in software on a personal computer or embedded system.[1] Functions that were typically carried out solely on hardware can now be performed by software that controls high speed signal processes. Its aim is to get code as "close to the antenna as possible" and turns a hardware problem into a software one.[2] This is helpful due to the lower entry barrier and the ability for hobbyists to test their ideas on

[1] Software Defined Radio Architectures, Systems, and Functions (Dillinger, Madani and Alonistioti, 2003)
[2] GNU Radio, (Chen and Chen, n.d.)

software. An advantage of using SDR is that you're able to achieve very high levels of performance. However, one disadvantage is that it can be difficult to write software to support different target platforms and may need to be redone.

3.1- FPGA, Digital Circuits & Logic:

Created by Xilinx in 1985, FPGAs are programmable silicon chips built from a large collection of programmable logic blocks.

FPGA designs use two basic types of logic: synchronous and combinatorial. Synchronous logic performs the read and write operations when a clock signal (a signal that oscillates between a high or low state, acting like a metronome) rises or falls. Combinatorial logic reads and writes depending on the speed of the signals being sent through gates or wires in the chip.

Synchronous logic uses flip-flops to hook onto an input value at each clock edge, guaranteeing that the output value will only change at the start/end of each clock cycle.

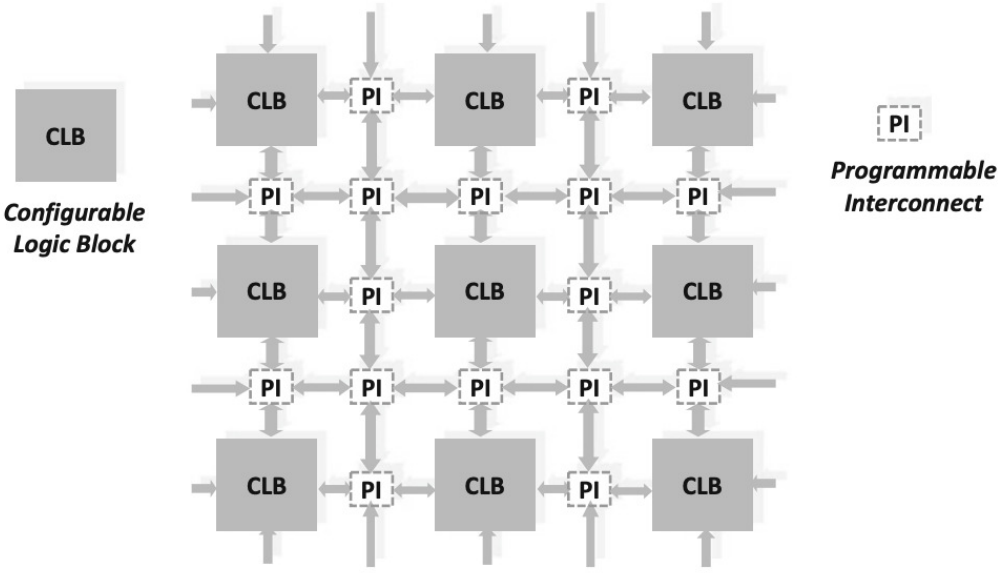

Fig 1.1 A general FPGA architecture

Modern FPGAs are no longer composed of simply an array of gates (i.e., LUTS) only. A LUT, or Lookup Table, is a table that determines what the output would be for any given input. Like a customised truth table, changing your inputs will change your output values. They are grouped with flip-flop registers (DFFs) and some carry logic in PLBs or CLBs. CLB's contains LUTS (used for combinatorial logic) and DFFs(used to store information)

Input A	Input B	Output C
0	0	0
0	1	0
1	0	0
1	1	1

Fig 1.2 AND Gate Lookup Table

3.1.1- ASICs and CPUs:

There are a few reasons why choosing an FPGA is the right choice in comparison to an ASIC or a CPU for this project. CPUs are perfect for general-purpose computations. It is a traditional sequential processor for general purpose applications and can do a slew of things. However, for this project we only need to accelerate DSP functions and not much else.

Ideally, an ASIC would be used. The custom integrated circuit would be fully optimised for the end application since it would be application specific. Additionally, due to the custom design being tailored towards the end application, it could be optimised for a combination of performance and power consumption.[3] However, the drawbacks of using an ASIC for this project are the long development time and high development cost ruling this option out entirely.

A good middle ground between performance and power consumption, price, and availability comes in the form of an FPGA. For this project, I will be using an Acorn CLE-215+, a cryptocurrency mining accelerator card repurposed as an FPGA for the project.

Entry level FPGA's emphasise lower power consumption, low logic density and low complexity per chip. [4] At the high end, they can include complex SoC parts that can be integrated with the FPGA's architecture. Complex tasks can be solved by software acceleration software via tools such as parallelisation and adaptation to the end application, providing a significant speed advantage in comparison to other processors.

[3] "FPGA vs CPU vs GPU vs Microcontroller: How Do They Fit into the Processing Jigsaw Puzzle?" (Arrow, 2018)
[4] "FPGA Basics: Architecture, Applications and Uses?" (Arrow, 2018)

Fig 1.3 Acorn CLE-215+

3.2- Digital Signal Processing:

Digital signal processing (DSP) is the use of computers or digital signal processors to perform a variety of signal operations. DSP can be found in a wide variety of places such as telecommunications, digital image processing. For our project, the signal we use is signal source in GNU Radio which generates a cosine waveform. Our project does involve some DSP math which is done on the FPGA

3.3 Kernel Modules:

For our FPGA to work as intended, we need to first load an FPGA gateware image is needed to be built and loaded onto the FPGA board. This is done with the use of OpenOCD, an Open-On-Chip Debugger which helps to provide "debugging, in-system programming and boundary-scan testing for embedded target devices." [5]. Using the command "*python3 -m litex_boards.targets.sqrl_acorn --with-pcie --build --load –driver",* this generates all the files needed for the driver. " This also loads the FPGA gateware image onto the board. Furthermore, the FPGA bitstream needs to be loaded. An FPGA bitstream is simply a file that contains the programming information for an FPGA. [6] It can contain a description of the hardware logic, routing and the initial values for the registers and the on-chip memory.

In the kernel directory of our server, we need to compile the kernel module using "*make"*. The next step in the process is cryptographically signing the kernel module so that it can work with Secure Boot/UEFI. Secure boot is a security process designed to protect a system against malicious code being loaded and executed early in the boot process. If malicious or invalid binary code is loaded while secure boot is enabled, the host will be alerted, and the system will refuse to boot the faulty binary code. [7]

Running "*dmesg | tail"* will allow us check the system log and to see if the module was successfully loaded and any status information that comes with it.

```
cardiff@server-fpga:~/build/sqrl_acorn/driver/user$ dmesg | tail
[  250.151941] litepcie 0000:01:00.0: enabling device (0000 -> 0002)
[  250.152122] litepcie 0000:01:00.0: Version LiteX SoC on Acorn CLE-101/215(+) 2021-10-26 14:39:13
[  250.152152] litepcie 0000:01:00.0: 1 MSI IRQs allocated.
[  250.152167] litepcie 0000:01:00.0: Creating /dev/litepcie0
[  250.152205] litepcie 0000:01:00.0: Creating /dev/litepcie1
```

Fig 1.5 Dmesg log

Using the "*lspci -tvv"* command, we can examine the all the PCI devices connected to our computer. Our device is successfully connected and working and shows up in the PCI slot as "Xilinx Corporation Device 7024".

```
cardiff@server-fpga:~/build/sqrl_acorn/driver/user$ lspci -tvv
-[0000:00]-+-00.0  Intel Corporation 4th Gen Core Processor DRAM Controller
           +-01.0-[01]----00.0  Xilinx Corporation Device 7024
```

Fig 1.6 lspci log

For the FPGA to be able to communicate with our computer, we need to load a kernel module onto the system to allow communication between the Acorn CLE-215+ and the computer. The kernel is a program that runs at the core of a computer's operating system. It sits between the hardware and the users' applications. When the system

[5] OpenOCD Manual (2021)
[6] "FPGA Bitstream" (Xilinx, 2018)
[7] "Take Control of Your PC with UEFI Secure Boot" (Paul G, 2015)

loads, the kernel is the first program that is loaded after the bootloader (the program responsible for booting up a computer). Its job is to be able to communicate with the applications and hardware (e.g., network cards, the CPU, a printer, an FPGA etc etc). For this communication, a kernel module is needed. Kernel modules are pieces of compiled binary code that can be loaded/unloaded into the kernel. This allows for extending the functionality of the kernel without the need to power cycle the system or recompiling the entire kernel.

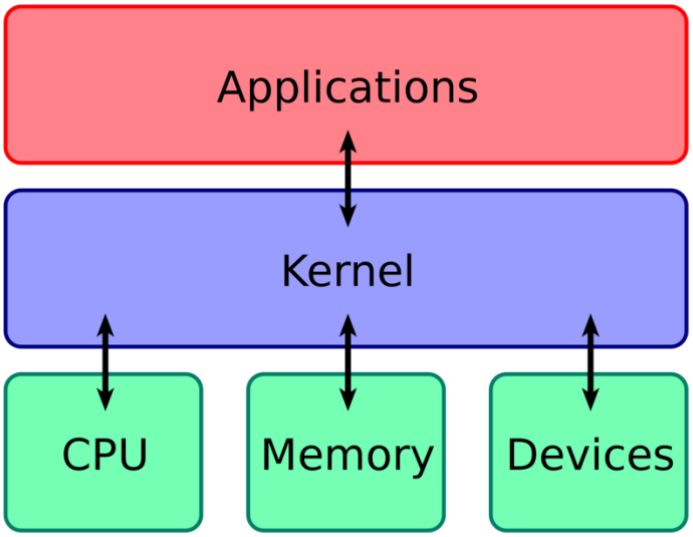

Fig 1.4 Diagram of kernel

3.4 - GNU Radio, LiteX and Alternatives:

GNU Radio is a software development framework that allows for the creation of signal processing blocks to implement software radios.[8] Instead of using traditional hardware, GNU Radio performs the signal processing in software. In the system, GNU Radio has pre-built blocks which allow for the processing to be carried out. GNU Radio has prebuilt in blocks such as filters, equalisers, vocoders, and many other blocks which are typically found in radio systems. These blocks can be connected using the GNU Radio which allows for easy manipulation of said blocks.

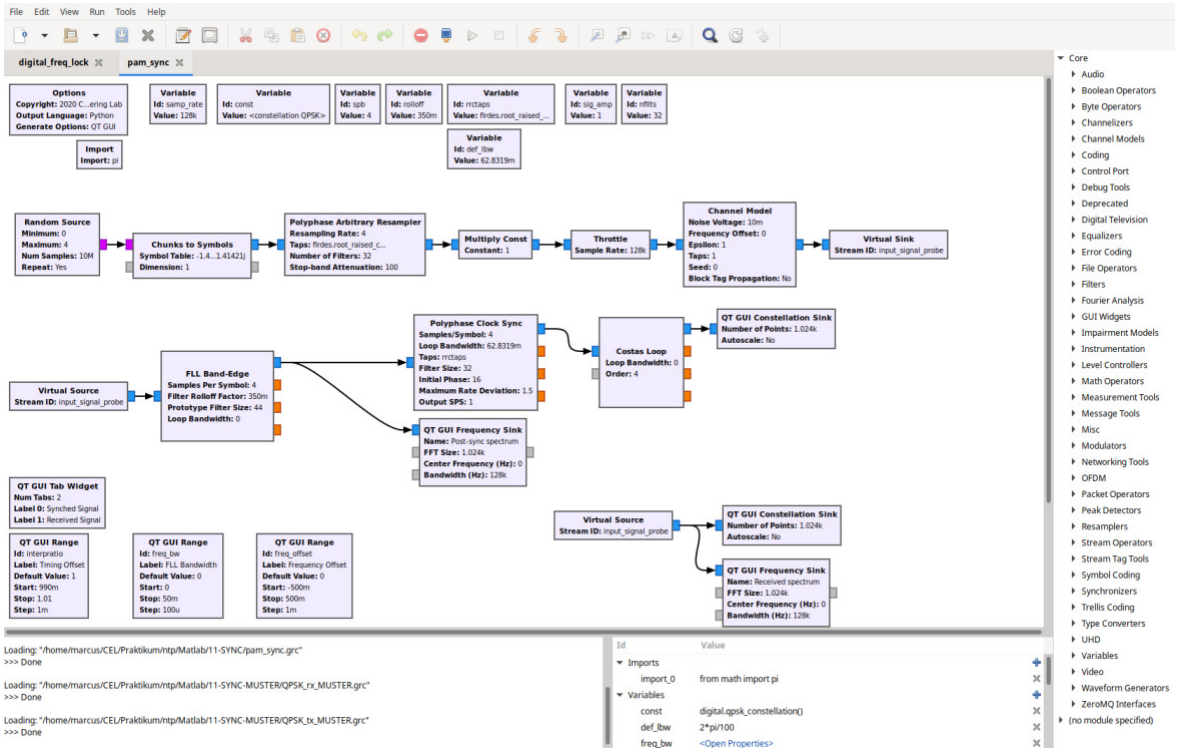

Fig 1.7 GNU Radio Companion

These blocks can be interconnected to create sophisticated software radios. It allows for easy-to-use reusable blocks and offers a large amount of scalability while providing an extensive library of standard algorithms.[9]

GNU Radio allows for the addition of functionality by writing code in Python or C/C++ using either creating an OOT or using an embedded block. Performance critical code, however, should be written in C/C++.

However, GNU Radio is not the only SDR framework on the market. MATLAB Simulink is a SDR that allows you to design and simulate your designs before moving onto actual hardware. Similar to GNU Radio and other SDR's, its primary interface is a graphical block diagramming tool, allowing you interconnect blocks to create complex designs. Simulink

[8] "What is GNU Radio?" - (GNU Radio, 2020)
[9] "Why would I want to use GNU Radio?" - (GNU Radio, 2020)

would not be entirely appropriate for this dissertation due to it mainly being used for modelling dynamical systems.

Another example is REDHAWK. REDHAWK is a SDR designed to "support the development, deployment, and management of real-time software radio applications" [10] Unlike GNU Radio, REDHAWK does not have such a large public following with a community to help troubleshoot issues that may arise. Furthermore, GNU Radio already has a large number of algorithms built in, saving the developers time

A SDR that builds on top GNU Radio is Amalthea. Amalthea is an "experimental SDR platform" and states that the main components of the Amalthea hardware are a Lattice ECP5 FPGA, an AT86RF215 radio transceiver, and a Microchip USB3343 USB2.0 PHY. It is currently provided as an OOT for GNU Radio and shares similar functionality.

Amalthea's Hybrid SDR is a toolkit for building SDR containing a mixture of software running on a general-purpose computer and gateware running an FPGA.[11] The crossing between gateware and software domains are handled by a custom-built USB device, LUNA.[12] Additionally, Amalthea builds the FPGA gateware image automatically using the blocks and connects in GNU Radio. It is a small prototype and is not a commercial product and is only usable with the Amalthea hardware.

Also building on top of GNU Radio is RFNoC (RF Network on Chip). Created by Ettus Research, RFNoC is similar to GNU Radio and LiteX integration, it allows for an FPGA to be integrated into the USRP (Universal Software Radio Peripheral) signal processing chain. However, the downside with this is that it can only be used with Ettus research products. For RFNoC, the FPGA gateware image has to be built separately but can be used with any of the Ettus Research radios. For this reason, LiteX was chosen for the fact that it can be used with any FPGA.

LiteX is an open-source framework that allows for efficient infrastructure to create FPGA cores(blocks)/SoCs to create fully fledged FPGA based systems. [13] LiteX will work on many different boards from many different vendors. And for this reason, we will be implementing LiteX into GNU Radio and extending its functionality mainly focusing on LitePCIe.

[10] REDHAWK (2021)
[11] "Introduction", (greatscottgadgets/amalthea, 2021)
[12] (GitHub - greatscottgadgets/luna: a USB multitool + nMigen framework for monitoring, hacking, and developing USB devices, 2021)
[13] LiteX, (enjoy-digital, 2021)

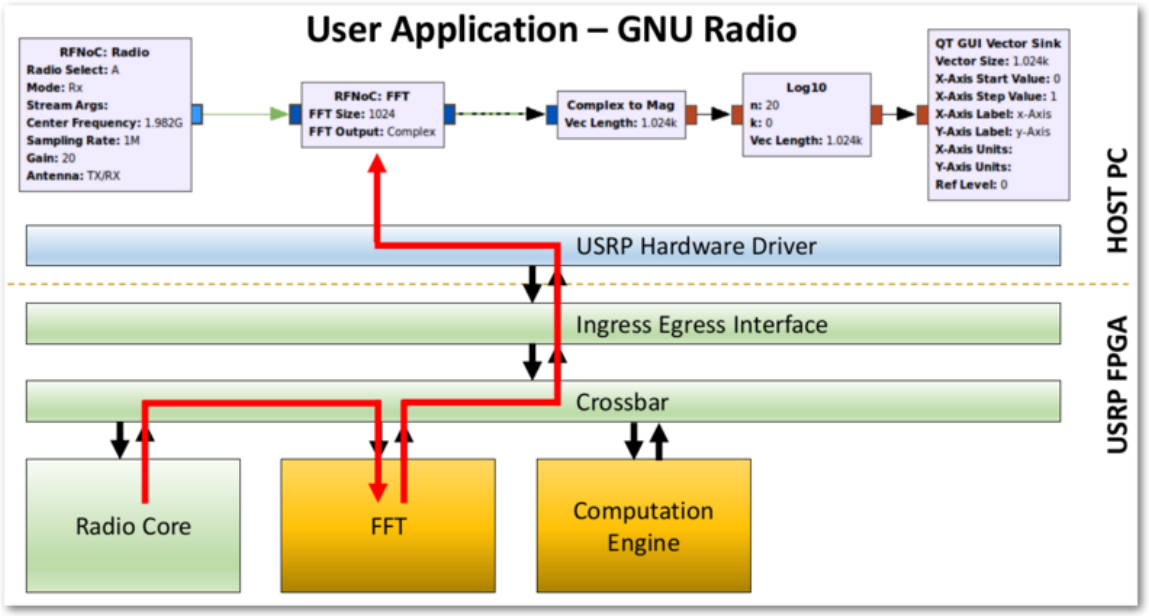

Fig 1.8 Basic data flow of RFNoC application

3.5 – PCIe & LitePCIe:

PCIe (Peripheral Component Interconnect Express) is a high-speed serial computer expansion bus standard which was designed to replace older and slower expansion slots such as PCI and AGP(Accelerated Graphics Port). In serial transmission, data flows in a specific order, bit by bit. In parallel transmission, multiple data bits are transmitted over multiple channels at the same time, meaning data can flow faster than using serial transmission.[14]

The problem with these older designs is that with the development of new technologies, they either became too slow or were not compatible with a wide range of products (e.g., the AGP was only compatible with graphics cards. Data is transmitted to and from PCIe slots in what are called lanes. PCIe x1 would indicate that there is one lane of data transmission. If you couldn't speed up the rate of data transmission in a singular lane, you could increase data transmission by adding lanes. PCIe x4 would be an increase of factor 4 lanes.

	x1	x4	x8	x16
PCIe 1.0	250MB/s	1GB/s	2GB/s	4GB/s
PCIe 2.0	500MB/s	2GB/s	4GB/s	8GB/s
PCIe 3.0	985MB/s	3.94GB/s	7.88GB/s	15.8GB/s
PCIe 4.0	1.97GB/s	7.88GB/s	15.8GB/s	31.5GB/s
*PCIe 5.0**	*3.94GB/s*	*15.8GB/s*	*31.5GB/s*	*63.0GB/s*

Fig 1.9 PCIe speeds

The Acorn CLE-215+ has been repurposed as an FPGA. It uses the Artix 7 200T FPGA which supports PCIe Generation 2 and has up to 4 lanes. The board has been fitted with a NVME PCIe x16 adapter to allow for greater data transmission speeds.

LitePCIe is a library implemented in LiteX which provides a configurable PCIe core which will be used for the data transfer from GNU Radio to LiteX. There are 3 important parts to the library. Firstly, It includes the userspace C library "*liblitepcie.c*" which C and C++ user programs can use to communicate through the kernel to the FPGA. Secondly, the LitePCIe library includes the kernel module that is used to interact with the PCIe core. And lastly, it contains the FPGA gateware implementing the actual hardware description of a PCIe core. This is often called an IP core (intellectual property core) when referring to FPGA designs.

[14] Data Transmission - Parallel vs Serial Transmission (Bin Ni, 2021)

3.6 – Operating System:

Operating systems are essential parts of any computer systems. The operating system controls the hardware and coordinates its use among the various application programs for the various users. [15] For this project, we will be using ubuntu Linux as our operating system of choice. We chose this for several reasons. Firstly, it's easy to install and get up and running. Secondly, GNU Radio is native to Linux so using any other operating system would make it harder to install than it needs to be. Thirdly, Linux is known for its stability making it the operating system of choice for our project. We also chose the ubuntu distribution over say, Arch or Gentoo, due to its user friendless and my past experiences using it.

4 – Problem Description:

The host computer should be able to create a GNU Radio flowgraph and implement the "LiteX Accelerator" block, our custom-made accelerator block, and have data transferred to and from the FPGA in a data loopback so information is read and then written to the FPGA.

In order to get this fully functional, we first need to get the FPGA first working as intended so we can later integrate it with GNU Radio. This is done by making sure we have a working FPGA bitstream loaded onto the device.[16] This was made more difficult due to the fact I was working completely remote for the entirety of this project, and it was all done over SSH with created an extra layer of obscurity, but this was difficulty was quickly mitigated with learning more about SSH and about the terminal.

Secondly, we needed to get GNU Radio interacting with the FPGA. GNU Radio and LiteX had to be able to communicate with each other for this project to work. Without this, we cannot get the FPGA embedded into GNU Radio to speed up the DSP functions in a way that is easy for the developer. It would have to be done manually which would be timely and complicated.

Thirdly, we need to create a GNU Radio OOT so that it would accept data transfer. Further difficulty arose here because this was intended to be done in Python but switched to C++ due to change of scope in the project and because working with C/C++ is a lot easier with LiteX and GNU Radio than it is in python(also due to the majority of people available to answer queries use C/C++). Our initial example should be very simplistic just to be able to show that the functionality works and giving us a foundation to build on top of and create more complex examples if wanted. For our example, it simply takes in a number of floats, and returns them back. The resulting flowgraph is a sine wave. At this point of time, I had done zero programming experience in C++ so getting to grips with the language and learning it while contributing to the dissertation was difficult at times.

[15] Operating Systems Concepts, pg. 4 (Silberschatz, 2021)
[16] "Bitstream Explained" (Shan, 2021)

5 - Approach & Application:

A considerable amount of time was spent researching GNU Radio, LiteX, the basics of C++ along with topics such as FPGA's and how they work. This information was gathered from the GNU Radio and LiteX wiki, the IRC chat where developers would answer queries, the internet, and books.

The first part of the approach was to get the FPGA bitstream loaded onto the FPGA. The approach to solving this problem was to get a working bitstream and build and load it onto the FPGA. Further difficulties that arose would be solved by getting help from the LiteX chat. The bitstream used, "sqrl_acorn_2021_07_29", was provided by Florent Kermarrec of Enjoy Digital, creator of LiteX. Using the command *"python3 -m litex_boards.targets.sqrl_acorn --with-pcie --driver --build –load",* this rebuilds and loads the FPGA image. The FPGA bitstream was loaded onto the device manually by Mr. Kozel using Vivado's Hardware Manager.

After this, our next target was to get GNU Radio interacting with LiteX. This would simply be done by including necessary header files from LiteX into GNU Radio. All header files came from the LitePCIe library of LiteX.

Lastly, we would need to create our own block in C++ to allow for data to be read and written. This would require extending "litepcie_util.c" located in the user directory of LitePCIe to allow for data loopback in GNU Radio.

GNU Radio has the "*gr::block"* implemented inside of it. It is the abstract base class for all 'terminal' processing blocks. [17] For example, the general_work block is called to perform the signal processing. It reads the input items and writes the output items. It is where the majority of the work in our program is done.

In the start function of our file "*litexgnu_impl.cc"*, our FPGA device is initialised and opened. This was done in the gr::block::start() function due to it being used to enable drivers for i/o devices.

```
bool litexgnu_impl::start()
{
    static char litepcie_device[1024];
    static int litepcie_device_num = 0; //Channel number

    snprintf(
        litepcie_device, sizeof(litepcie_device), "/dev/litepcie%d", litepcie_device_num);


    fds.fd = open(litepcie_device, O_RDWR | O_CLOEXEC);
    fds.events = POLLIN | POLLOUT;
    if (fds.fd < 0) {
        fprintf(stderr, "Could not init driver\n");
        exit(1);
    }
```

Fig 2.1 Start function

[17] GNU Radio Manual and C++ API Reference: GR::BLOCK CLASS REFERENCE (GNU Radio Foundation, 2021)

Our FPGA is treated as a file and is opened as a file descriptor, *"fds.fd"* with 2 flags, 0_RDWR and 0_CLOEXEC. 0_RDWR is set to make sure the FPGA can only be read or written onto and 0_CLOEXEC enables the close on execution flag. We poll the device to make sure it is ready to perform a task. If the device is greater than 0 (if a device is ready and is able to be used), it opens, else it prints out the error message "*Could not init driver",* and subsequently exits. Similar to the *gr::block::start(),* there is a *gr::block::stop()* which disables the drivers. It is in here where the device is closed. Originally, this was in a function called in the now defunct "*info"* which would open the FPGA and read the FPGA identification. This function was then called in the general_work block. However, using the start function improved readability and allowed for easier refactoring.

Further down in the start function is the FPGA identification. This part of the program loops and identifies uses "*litepcie_readl"* to print out the FPGA device and date and time that is stored in "*CSR_IDENTIFIER_MEM_BASE".* This is a memory address. The LiteX SoC has an internal memory that can be read, and sometimes written. This address holds the ID string. The loop in figure 2.2 reads one character at a time.

The last part of the start function is the dma loopback. We enable dma loopback on the device to allow data to be sent from the host to the device and back to the host again in a loop continuously.

```
unsigned char fpga_identification[256];

for (int i = 0; i < 256; i++)
    fpga_identification[i] = litepcie_readl(fds.fd, CSR_IDENTIFIER_MEM_BASE + 4 * i);
printf("FPGA identification: %s\n", fpga_identification);
```

Fig 2.2 CSR Identifier Mem Base loop

There are 2 important limits that need to be calculated to avoid overflow. The first is the sum of the DMA buffer sizes. The second is the number of available items in GNU Radio. Each DMA buffer can hold 8192 bytes of data. The number of available items in GNU Radio is calculated by the DMA Buffer total size (DMA count * DMA Buffer size = 2097152 bytes) divided by the size of input_type in bytes. The sum of the DMA buffer count is 2097152 bytes.

In the general_work block, it contains the write event. This is the code that shows how data is written onto the device. max_items_write is the DMA_BUFFER_TOTAL_SIZE divided by the sizeof(input_type). The DMA_BUFFER_TOTAL_SIZE is defined as the DMA Buffer Count (256) * the DMA_BUFFER_SIZE (8192) making the DMA buffer total size 2097152 bytes. The sizeof(input_type) returns the size of the input_type in bytes. n_dma_blocks is the number of DMA blocks. This is calculated by multiplying the number of input items by the DMA_BUFFER_SIZE and dividing that by the size of the input_type in bytes. n_dma_items is the number of DMA blocks multiplied by the DMA buffer size divided by the size of the input_type in bytes. n_write_items are the items that are written to the device. It uses std::min to calculate the minimum number between the max_items_write and the n_dma_items).

bytes_written uses “write” to write to the device. The first parameter is the FPGA device, “*fds.fd”*. The 2nd parameter is a void pointer which points to the variable “*in”* which is pointer to a pointer of the memory location of the input items. If the bytes_written are not equal to the n_write_items * size of the input_type (which should give us the maximum number of bytes available), an error message should be printed out.

```
/* write event */
if (fds.revents & POLLOUT) {
    int max_items_write = DMA_BUFFER_TOTAL_SIZE / sizeof(input_type);
    int n_dma_blocks = (ninput_items[0] * sizeof(input_type)) / DMA_BUFFER_SIZE;
    int n_dma_items = n_dma_blocks * DMA_BUFFER_SIZE / sizeof(input_type);
    int n_write_items = std::min(max_items_write, n_dma_items);


    bytes_written = write(fds.fd, (void*)in[0], n_write_items * sizeof(input_type));

    // if (bytes_written != n_write_items * sizeof(input_type)) {
    //     std::cout << "Error: Max bytes already written" << '\n';
    // }   - This does not need to be printed everytime but still good to keep in
    consumed_items = bytes_written / sizeof(input_type);

    // std::cout << "Bytes written: " << bytes_written << '\n';
}
```

Fig 2.3 Write event code

This has been commented out since this functionality is not need in this stage, however, it has not been removed because the code is still useful. If further development was done with the reading/writing, this could be uncommented out. The number of consumed items is the bytes_written divided by the size of the input type. At the end of the general_work block, the block can report how many items were consumed on each input stream using consume() or consume_each().

Similar to the write event, is the read event. The main mechanics of the code are the same as that of the write event. Instead, the read event uses POLLIN. Furthermore, for the bytes_read it uses a void pointer which points to the variable "*out*" which is pointer to a pointer of the memory location of the output items.

```
/* read event */
if (fds.revents & POLLIN) {
    int max_items_read = DMA_BUFFER_TOTAL_SIZE / sizeof(output_type);
    int n_read_items = std::min(max_items_read, noutput_items);

    // std::cout << "Reading items: " << n_read_items << '\n';
    bytes_read = read(fds.fd, (void*)out[0], n_read_items * sizeof(output_type));

    // if (bytes_read != n_read_items * sizeof(output_type)) {
    //     std::cout << "Error: Max bytes already read" << '\n';
    // } - Again, this does not need to be printed everytime but still good to keep in
    created_items = bytes_read / sizeof(output_type);
    // std::cout << "Items read: " << created_items << '\n';
}
```

Fig 2.4 Read event code

Another important part of the code is the statistics. This measures the rate of data transfer in Gigabytes per second (Gbps). The duration is the time in milliseconds (implementation located in *liblitepcie*.c) minus last_time (implementation located in *litexgnu_impl.h*). The speed is the difference between the reader software count(reader_sw_count) and the last reader software count (which is initialised to 0) multiplied by the DMA buffer size * BITS_PER_BYTE (8 bits in a Byte) divided by the duration * 1e6 (number of ms/s).

```
/* statistics */
duration = get_time_ms() - last_time;
if (duration > 200) {
    double speed = (double)(reader_sw_count - reader_sw_count_last) *
                   DMA_BUFFER_SIZE * BITS_PER_BYTE / ((double)duration * 1e6);
    if (work_iteration % 10 == 0)
        printf("\e[1mDMA_SPEED(Gbps) TX_BUFFERS RX_BUFFERS  DIFF  ERRORS\e[0m\n");
    work_iteration++;
    printf("%14.2lf %10" PRIu64 " %10" PRIu64 " %6" PRIu64 " %7u\n",
           speed,
           reader_sw_count,
           writer_sw_count,
           reader_sw_count - writer_sw_count,
           errors);
    errors = 0;
    last_time = get_time_ms();
    reader_sw_count_last = reader_sw_count;
}
```

Fig 2.5 Statistics code

The variables needed for this to run are located in the "*litexgnu_impl.h*" file. They are private instance variables of the "*litexgnu_impl*" class. Each litexgnu_implcc object gets its own private set of these variables.

6 - Testing, Results & Analysis:

For testing the transfer speeds, we created a very simple GNU Radio flowgraph. It takes in a signal source with a singular output, a sample rate of 32k and, a waveform of cosine. It is connected to our 'LiteX Accelerator' block which is then connected to a throttle block set to 32k and a probe rate. The throttle block is disabled as the sample rate is set to 32k; however, the rate of data transfer is more than of 100x of that. If re-enabled, it is there to make sure the average rate does not exceed a certain samples per second. The throttle block is connected to a QT GUI Time Sink to visualise this.

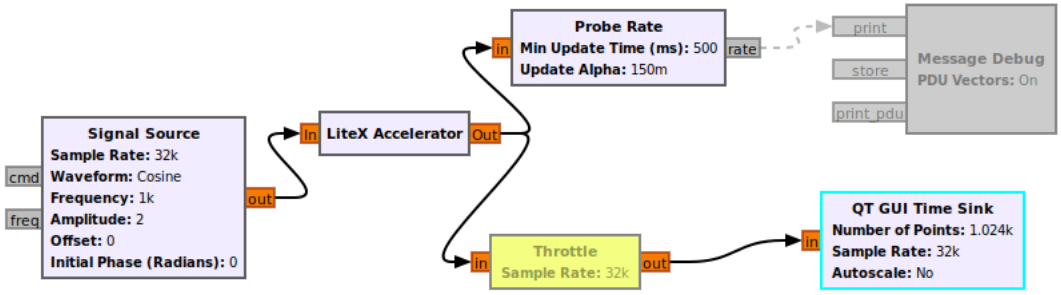

Fig 2.6 Demo flowgraph

The probe rate block is connected to measure throughput. This probe_rate block is connected to the "*print*" input of the disabled message debug block. The message debug, when enabled, prints out the current rate of data transfer and the rate average. When the flowgraph is run, it should produce a sine wave flowgraph if successful.

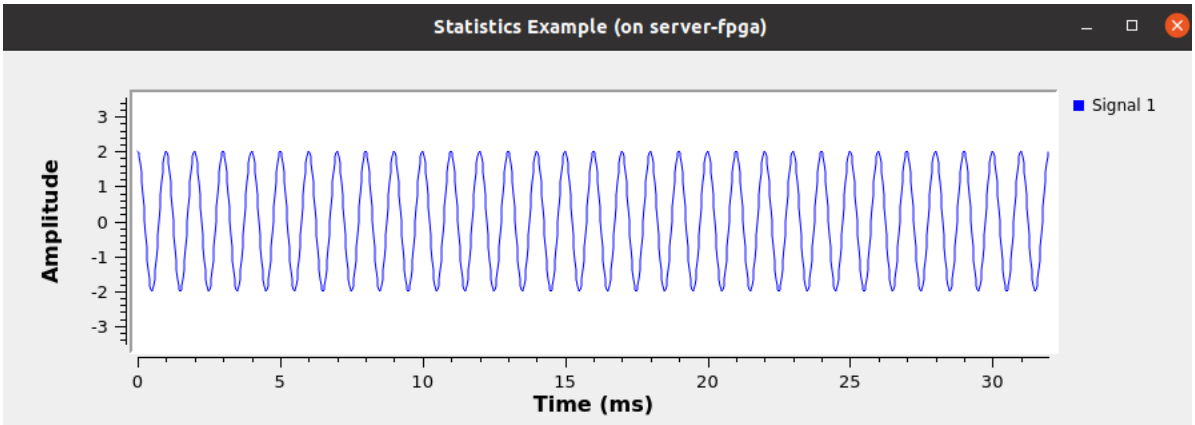

Fig 2.7 Sine wave output

Running the flowgraph gives us the correct output that we expected. When looking at the console, it prints out the FPGA identification along with the DMA Speed, TX_BUFFERS, RX_BUFFERS, the difference between the two and, the errors(although, the errors has been initialised to 0 so this will always be zero). The first iteration gives us 0 for everything. This is simply because, it hasn't loaded properly and should be ignored. In subsequent iterations, we can see we get a DMA speed from around 8.49 – 8.52Gbps. The highest theoretical throughput is 16Gbps. A speed increase may arise from further optimising the code or by

using a better FPGA. Furthermore, recent changes in GNU Radio itself might help to see a speed boost. Recently, GNU Radio introduced the concept of “Custom Buffers”. This allows DMA transfers to be controlled by GNU Radio runtime rather than the LiteX accelerator block. However, this feature was not available at the start of the project and may or may not speed up the data transfer rate. Additionally, increasing the buffering in the LitePCIe implementation might increase the speed , however both things have not been tested.

As we can see, there is a difference between the TX_BUFFER and the RX_BUFFER. Ideally, the difference should be 0, however, the current average is around 130.

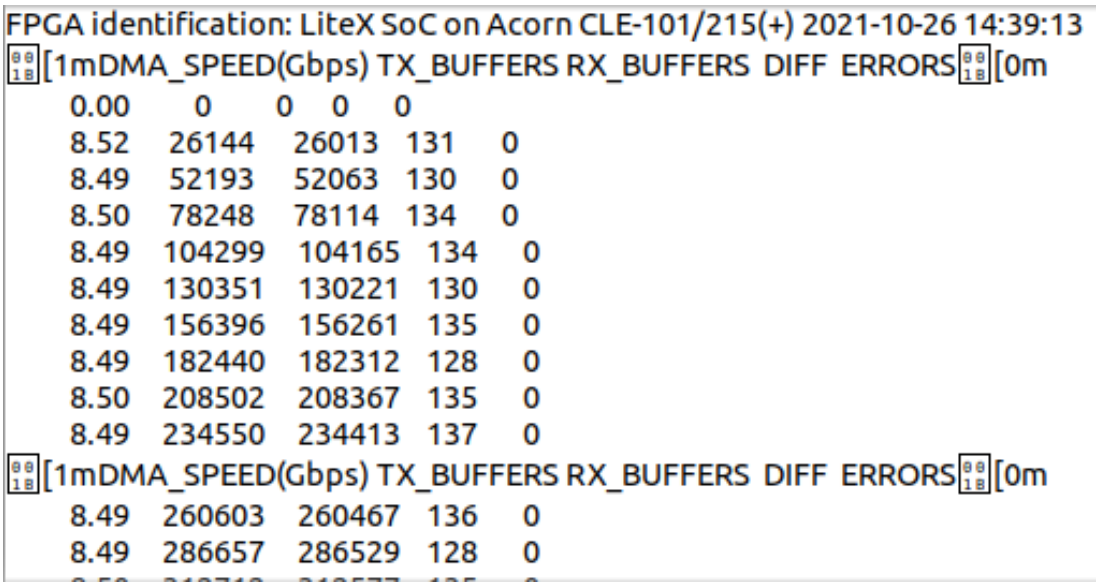

Fig 2.8 DMA speed in GNU Radio console log

When running the DMA test, we can see we are achieving identical speeds in comparison to the data transfer in GNU Radio.

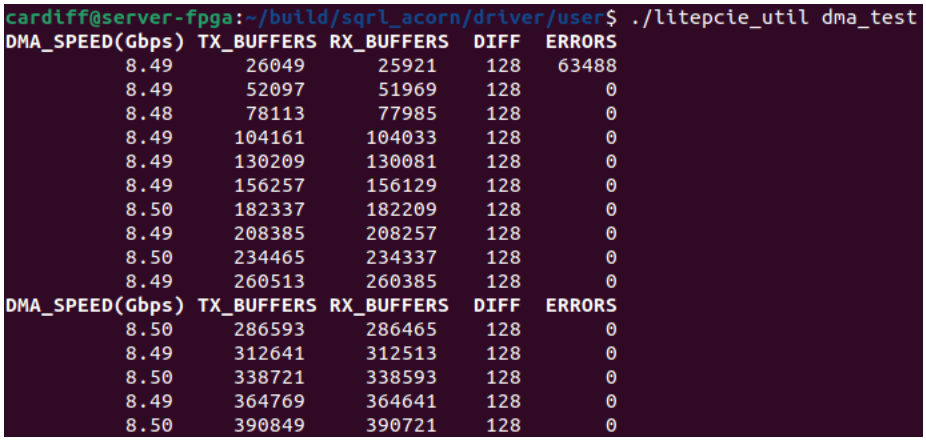

Fig 2.9 DMA speed console log

7 – Conclusion:

Based on the results given to us by testing, this dissertation could be regarded as a success. Although not all major objectives set out in the beginning were met, due to the high-level base knowledge required for such a dissertation, I hope leniency would be applied. This dissertation shows that LiteX can be implemented into GNU Radio and can be used for data transfer in a loopback at quite high speeds. Another additional step would be to explore why the TX_BUFFER and RX_BUFFERs aren't aligning and why there is an average difference of 130 on each iteration. An additional next step would be fleshing out the demo more and using multiple input channels. If there was time for future work, it would be beneficial, to see how the use of custom buffers could potentially help with a speed increase. Playing around with the LitePCIe buffer rate would also be on the next steps. The addition of multiple channels in GNU radio is also one I would explore next.

Additionally, possibly separating the read and write functions into LitePCIe source and sink blocks could help with the increase of speed since each block would run its own thread. Again, this would need to be tested.

8 - Reflection:

This dissertation has by far been the most difficult and complex project I have worked on. However, it has also been the most interesting and rewarding thing I have been a part of. I have learnt things that weren't taught on the master's course and have been able to contribute to an open-source project that does useful things. It has been hard work, but it has definitely stretched me.

I have learnt many important things during the span of this project. Time management being one of the most important. The ability to learn complex topics in a timely manner has also been very important.

One of the most important skills I have learnt is how to learn from my mistakes. A project that required me to learn essentially everything from the ground up was always going to have problems. However, the resilience needed to fail and make mistakes but to continue pushing along was something I don't think I had at the start of this project.

Accelerating DSP functions for GNU Radio by implementing them on an FPGA-based PCIe/Thunderbolt co-processor

References:

- Dillinger, M., Madani, K. and Alonistioti, N., 2003. Software Defined Radio: Architectures, Systems and Functions. [online] O'Reilly Online Learning. Available at: https://learning.oreilly.com/library/view/software-defined-radio/9780470851647/11_chapter001.html#ch001-sec001
- Chen, Z. and Chen, K., n.d. GNU Radio. [online] Wu.ece.ufl.edu. Available at: <http://www.wu.ece.ufl.edu/projects/softwareRadio/>
- Arrow. 2018. FPGA vs CPU vs GPU vs Microcontroller: How Do They Fit into the Processing Jigsaw Puzzle?. [online] Available at: <https://www.arrow.com/en/research-and-events/articles/fpga-vs-cpu-vs-gpu-vs-microcontroller>
- Arrow. 2018. FPGA Basics: Architecture, Applications and Uses [online] Available at: <https://www.arrow.com/en/research-and-events/articles/fpga-basics-architecture-applications-and-uses >
- Openocd.org. 2021. [online] Available at: <https://openocd.org/doc/pdf/openocd.pdf>
- Xilinx.com. 2018. FPGA Bitstream. [online] Available at: <https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html>
- Paul, G., 2015. Take Control of Your PC with UEFI Secure Boot | Linux Journal. [online] Linuxjournal.com. Available at: <https://www.linuxjournal.com/content/take-control-your-pc-uefi-secure-boot>
- Wiki.gnuradio.org. 2020. What is GNU Radio? - GNU Radio. [online] Available at: <https://wiki.gnuradio.org/index.php/What_is_GNU_Radio%3F>
- Wiki.gnuradio.org. 2020. Why would I want to use GNU Radio? [online] Available at: <https://wiki.gnuradio.org/index.php/What_is_GNU_Radio%3F>
- Redhawksdr.org. n.d. Home :: REDHAWK. [online] Available at: <https://redhawksdr.org/>
- GitHub. 2021. amalthea/intro.rst at main · greatscottgadgets/amalthea. [online] Available at: <https://github.com/greatscottgadgets/amalthea/blob/main/docs/intro.rst>
- GitHub. 2021. GitHub - greatscottgadgets/luna: a USB multitool + nMigen framework for monitoring, hacking, and developing USB devices. [online] Available at: <https://github.com/greatscottgadgets/luna>
- GitHub. 2021. GitHub - enjoy-digital/litex: Build your hardware, easily!. [online] Available at: <https://github.com/enjoy-digital/litex>
- Ni, Bin., n.d. Data Transmission - Parallel vs Serial Transmission. [online] Quantil.com. Available at: <https://www.quantil.com/content-delivery-insights/content-acceleration/data-transmission/>
- Silberschatz, A., 2021. OPERATING SYSTEM CONCEPTS. [S.l.]: JOHN WILEY, p.4.
- Shan, Y., 2021. Bitstream Explained - Yizhou Shan's Home Page. [online] Lastweek.io. Available at: <http://lastweek.io/fpga/bitstream/>
- Gnuradio.org. n.d. *GNU Radio Manual and C++ API Reference: gr::block Class Reference*. [online] Available at: <https://www.gnuradio.org/doc/doxygen/classgr_1_1block.html>