Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Radio interference monitoring with low-end AI-hardware

Olivier Visser
s3329038

Supervisors:
Prof. dr. Rob van Nieuwpoort & Dr. Chris Broekema

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

27/3/2025

Abstract

Radio telescopes are essential tools for studying radio-frequency emissions from space, yet they are highly affected by interference from man-made and natural radio signals. This project explores the use of low-cost, low-power hardware, specifically the Google Coral Edge TPU paired with an USB-based software-defined radio (SDR), to monitor Radio Frequency Interference (RFI). Unlike traditional applications of the Coral TPU for machine learning tasks, this project uniquely repurposes the hardware for non-AI computations, focusing on implementing the Fast Fourier Transform (FFT), a critical signal processing step, to evaluate its suitability for such tasks.

While the Coral excels in energy efficiency, the project faced challenges with limited support, outdated documentation and hardware constraints. Despite these issues, experiments demonstrated that with careful batching and optimization, the Coral can keep up with SDR data streams, offering an efficient solution for low-power, real-time signal processing. The results suggest that the Coral TPU is effective for specific signal processing tasks in low-power applications. However, to fully realize its potential in scientific applications, further developments, such as the addition of more advanced signal processing tasks or a general-purpose computing framework, are needed.

Contents

1 Introduction **1**

2 Background **1**
- 2.1 LOFAR . . . 1
- 2.2 RFI . . . 2
- 2.3 Coral Edge TPU . . . 3
- 2.4 RTL-SDR . . . 4

3 Related work **5**
- 3.1 General purpose computing on edge TPUs . . . 5
- 3.2 GPUs . . . 6
- 3.3 TPUs . . . 6
- 3.4 DPUs . . . 6

4 Design **7**
- 4.1 Gathering data . . . 7
- 4.2 Holding data . . . 7
- 4.3 Preprocessing . . . 8
- 4.4 Model Invocation . . . 8
- 4.5 Outputting Data . . . 8

5 Implementation **9**
- 5.1 Folding method . . . 10

6 Experiments **13**
- 6.1 Accuracy . . . 14
- 6.2 Performance . . . 15

7 Discussion **17**

8 Conclusions **18**

9 Future work **18**

References **21**

1 Introduction

A radio telescope is an astronomical instrument to detect and study radio-frequency emissions from beyond the earth's atmosphere. Unlike optical telescopes that observe visible light, radio telescopes are sensitive to radio waves. Radio Frequency Interference (RFI)[1] are the unwanted radio signals that may disturb the telescope, often described as noise from sources that emit strong electromagnetic signals, which 'deafens' the radio telescope and decreases the ability to pick up much weaker signals from far away in the galaxy. In most cases telescopes are built in rural places to reduce the effects of man-made RFI, however that may not always be possible and in those instances it is especially important to monitor the RFI and correct the radio telescope accordingly. RFI monitoring is generally done by specialized software, but that comes at the cost of reduced signal-to-noise ratio.

This project proposes the use of cheap and abundantly available USB-based radio receivers, originally designed as DVB-T receivers, combined with the powerful but energy efficient Google Coral Edge TPU to collect and process radio frequency data as proposed by Chris Broekema (ASTRON)[1]. While a radio telescope's full processing pipeline consists of many stages, this project will focus solely on implementing the FFT (Fast Fourier Transform)[2] function, one of the earliest and most critical stages. The aim of this project is not to build the entire pipeline, but to demonstrate whether the Coral TPU can effectively handle this specific part, which would suggest its potential for the other stages as well. The FFT function is also a widely used function with many hardware-optimized implementations available, making it a good benchmark for comparing performance across different systems.

The following research questions are addressed to validate this approach:

1. How effective are existing TPU solutions in performing signal processing tasks?
2. How can the workload of the SDR and TPU be distributed evenly?
3. Is the Coral Edge TPU a suitable accelerator for signal processing tasks?

2 Background

In this chapter, we will describe the necessary background required for the remaining chapters of this thesis.

2.1 LOFAR

The Netherlands Institute for Radio Astronomy (ASTRON)[2] operates the LOFAR (Low-Frequency Array)[3], a radio telescope. Unlike traditional dish-based radio telescopes, LOFAR consists of a large, distributed network of approximately 25,000 small antennas spread over multiple European countries, with its core array, shown in figure 1, located in the Netherlands.

[1] https://www.chrisbroekema.eu/student-projects/
[2] https://www.astron.nl

These antennas are connected to a central processing facility that combines and processes the signals into a useful dataset, allowing for highly sensitive observations of the universe.
LOFAR operates at low radio frequencies, between 10 MHz and 240 MHz. This range enables researchers to explore phenomena that are otherwise difficult to study, such as the formation and evolution of galaxies and detecting transient astronomical events, including pulsar emissions, fast radio bursts (FRBs).

Figure 1: LOFAR Superterp

A significant challenge posed by LOFAR's design is the vast amount of data generated by its antennas. Each observation produces immense datasets, requiring complex algorithms and substantial computational power to process. This leads to issues such as difficulties in transmitting and storing petabytes of data across the network, computational bottlenecks when processing high-resolution observations in real-time and challenges in mitigating radio-frequency interference from terrestrial sources, which is particularly problematic at these low frequencies. This research could pave the way for a distributed network of low-power, standalone RFI measuring stations, helping to streamline RFI processing for LOFAR.

2.2 RFI

Radio Frequency Interference (RFI) can originate from a wide range of sources, with the most common being man-made signals such as FM radio, television broadcasts, mobile phone signals, and satellites. These artificial signals are often much stronger than the faint radio waves coming from distant celestial objects, making it difficult for radio telescopes to pick up the much weaker signal of intrest. In addition to human-made interference, natural sources also contribute to RFI. Lightning strikes and other weather-related disturbances can create strong electromagnetic signals, and galactic background noise, like radiation from the sun or other celestial bodies, adds to the problem. Together, these signals can 'deafen' the radio telescope, blocking out the faint, important signals from distant stars, galaxies, and other cosmic sources.

To mitigate RFI, radio telescopes are often placed in remote areas, away from human activity. Additionally, advanced techniques like filtering and signal processing[4] are used to separate unwanted interference from the valuable astronomical data. However, as RFI sources continue to increase, active monitoring and real-time correction methods are becoming more essential to preserve the integrity of radio observations.

2.3 Coral Edge TPU

The Coral Edge TPU, shown in figure 2, is typically used for machine learning (ML) inference tasks[5], such as image recognition, object detection, and natural language processing, in low-power embedded systems. It is designed to efficiently execute pre-trained models. The hardware excels at handling matrix operations, such as those required for deep learning tasks, making it well-suited for applications where real-time inference is needed, like in smart cameras and IoT devices
However, using the Coral TPU in this project diverges from its standard use case. Instead of using the TPU for ML inference, this project focuses on leveraging its matrix operation capabilities to accelerate signal processing tasks, specifically the Fourier transform. While matrix operations are common to both machine learning and signal processing, TPUs are not typically designed for general-purpose computing[6] or scientific workloads like a Graphics Processing Unit (GPU) or Central Processing Unit (CPU) might be. Additionally, creating a custom FFT model on the TPU comes with several restrictions[3]. For instance, models must be quantized to a signed 8-bit integer (INT8) format, where the entire model is scaled down from high-precision floating-point values to smaller INT8 values. This process can introduce some precision loss compared to the higher-precision floating-point formats commonly used in scientific computing. However, it significantly improves efficiency by enabling faster inference while reducing power consumption.

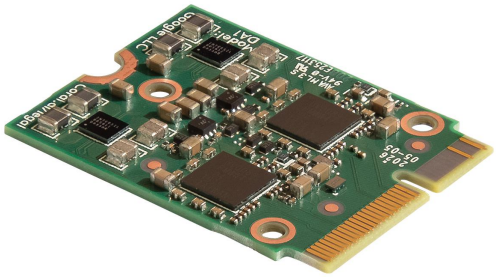


Figure 2: Coral M.2 Accelerator

The process of creating models for the Coral involves utilizing TensorFlow Lite[4] to convert high-level models into a format that the Edge TPU can execute. This conversion can be quite difficult due to the strict version matching needed across various dependencies, such as TensorFlow and the Coral libraries. Furthermore, optimizing the model's operations to conform to the TPU's constraints adds another layer of complexity. These factors combined can make the development process less straightforward and more time-consuming than more conventional methods.
Some of the Coral's constraints include its restriction to three-dimensional tensors and its limited support for the full range of TensorFlow Lite operations, which reduces flexibility when handling complex data structures and computations. This limitation is especially problematic for tasks like batched FFT calculations, which often require more advanced tensor manipulations.

A key benefit of the Edge TPU is it is performance per watt and cost-effectiveness, making it particularly suitable for applications like remote measurement stations where power consumption is a critical factor. Figure 3 illustrates the performance-to-power ratio, comparing the Coral TPU to the NVIDIA GTX 1080 GPU used in our testing.

[3] https://coral.ai/docs/edgetpu/models-intro/#model-requirements
[4] https://www.tensorflow.org/lite/

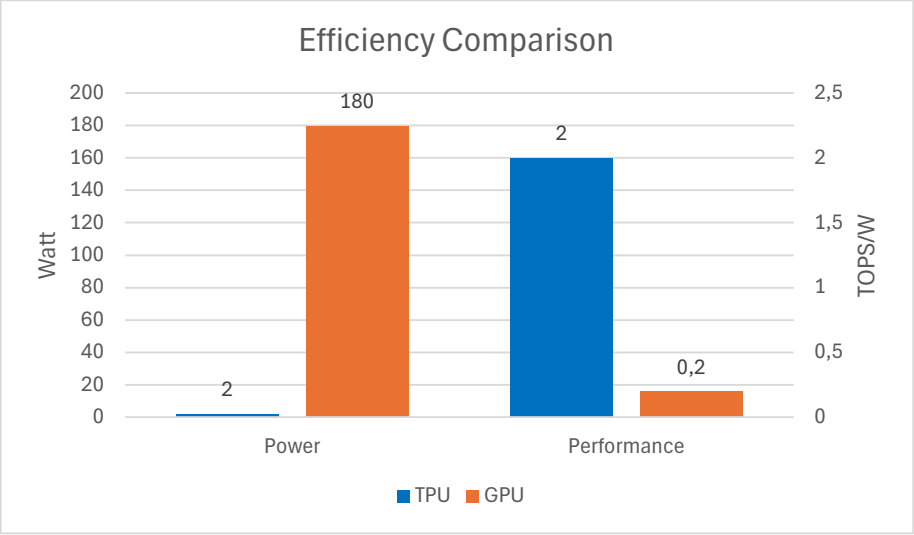


Figure 3: Comparison of maximum power consumption and operations per watt between the Coral Edge TPU and the NVIDIA GTX 1080 GPU.

As shown in the graph, the TPU significantly outperforms in both metrics, while costing only a fifteenth of the price. It is important to note that power and performance metrics are challenging to measure precisely, so these results should be taken as rough estimates for comparison purposes. According to Google[5], the Coral TPU delivers 4 TOPS (trillion operations per second) while consuming only 2 watts of power. For the GPU[6], we use its TDP (thermal design power) of 180 watts as an estimate of maximum power usage, combined with its rated 8.9 TFLOPS (trillion floating-point operations per second), which roughly translates to 35.6 TOPS.

2.4 RTL-SDR

The RTL-SDR[7], shown in figure 4, is commonly used as a low-cost software-defined radio (SDR) for applications like receiving FM and AM radio signals, weather satellite transmissions and even air traffic control communications. It is favored by hobbyists and researchers alike for its affordability and versatility, able to tune into a wide range of frequencies, from 500 kHz to 1.75 GHz. Normally, the RTL-SDR is not employed in professional-grade radio astronomy due to its limited sensitivity and dynamic range compared to specialized radio telescope receivers.

Figure 4: RTL-SDR V3

[5] https://coral.ai/docs/edgetpu/benchmarks/
[6] https://www.nvidia.com/en-ph/geforce/products/10series/geforce-gtx-1080/

In this project however, the RTL-SDR is specifically used for monitoring RFI, rather than capturing the weak signals from deep space typically observed in radio astronomy. The SDR's role is to pick up strong, unwanted RFI sources that interfere with sensitive astronomical observations. These loud RFI signals are later corrected from the high-quality data captured by the telescope, helping to clean up the final astronomical data.
However, there are challenges with this setup, such as the TPU's quantization process and real-time processing limits, which can cause dropped samples if the RTL-SDR's data stream exceeds the TPU's handling capacity.

3 Related work

3.1 General purpose computing on edge TPUs

GPTPU[8] was a project led by two researchers at the University of California, Riverside, that explores general-purpose computing using Google's Edge TPU, as suggested by the abbreviation. They claim a speed-up of 2.46x by leveraging the TPU's matrix processing units (MXUs) compared to a single high-end CPU core. In the paper they introduce, OpenCtpu, an edge TPU framework that enables developers to write programs that define TPU tasks and manage data exchanges. While this initially seemed ideal for the project, it ultimately did not meet expectations.

The project had several major flaws, the most significant being the lack of proper documentation. Many build steps were either missing or poorly explained, making it difficult to compile the library, in which we ultimately never succeeded. In addition to the insufficient documentation, there were numerous bugs and errors in the makefiles, such as the use of absolute paths and references to files that didn't exist in the repository. The code itself was disorganized, with some files containing more commented-out sections than actual functional code. In our opinion, the coding standards were far too low to use this project reliably in any serious application.

We attempted to refactor parts of the code and improve the build process, but with no success. A key issue was the unknown version of TensorFlow used for precompiling the templates. The Coral has strict requirements regarding version compatibility across the various libraries and programs needed and the TPU's vague or nonexistent error messages only complicated matters. After multiple failed attempts, we decided to abandon this project and develop our own implementation, though we used GPTPU's structure as a reference.

Additionally, the way GPTPU is described in the paper feels somewhat misleading, as key limitations are left out, such as the need for quantization, a limited instruction set and the three-dimensional constraint. These restrictions significantly undermine its claim of being a general-purpose solution. However, since we were unable to run the code, our conclusions are based on static analysis rather than actual testing.

3.2 GPUs

The GPU is the most commonly used device for large-scale computations on computers. Powerful GPUs are readily available to consumers and are widely used for tasks such as graphics rendering, AI training and video editing. Due to their popularity, extensive software support is available, with manufacturers developing their own general-purpose computing frameworks, such as NVIDIA's CUDA[9], to help developers easily leverage the GPU's processing power.
For this project, the availability of signal processing libraries is particularly relevant. Well-established libraries like FFTW[10] provide efficient FFT implementations, while numerous other specialized libraries are available for a wide range of signal processing applications[11]. With strong software support and consistently high performance, GPUs remain a reliable choice for intensive computational tasks.

3.3 TPUs

Tensor Processing Units (TPUs) are custom hardware accelerators developed by Google, designed to handle large-scale machine learning tasks. Unlike GPUs, which are general-purpose processors, TPUs are built specifically for working with tensors, multi-dimensional arrays of numbers used in deep learning. This makes TPUs extremely fast and efficient for AI applications like neural network training and inference[12]. Google provides software tools like TensorFlow's XLA compiler[7] and the JAX framework[8] to help developers optimize models for TPUs. However, while TPUs excel in AI tasks, they are not commonly used for general computing or signal processing. Unlike GPUs, which have well-supported libraries for tasks like FFTs, TPUs currently lack dedicated tools for these operations. However, implementations for such tasks are actively being researched[13].
Despite their efficiency in AI workloads, TPUs are less flexible than GPUs for a broader range of applications. They are best suited for machine learning tasks where power efficiency and speed are critical, but their limited support for general computing makes them less practical for tasks outside AI.

3.4 DPUs

Data Processing Units (DPUs) are specialized hardware accelerators designed to offload networking, storage, security, and certain computational tasks from CPUs, improving efficiency in data centers and high-performance computing environments. Unlike GPUs, which specialize in parallel computation for AI and graphics, DPUs, such as NVIDIA's BlueField[9], focus on data movement and processing, reducing CPU overhead in applications like cloud computing.
DPUs combine multi-core processors, high-speed networking and programmable accelerators to create efficient data processing pipelines[14]. This makes them well-suited for a signal processing pipeline. Unfortuanlly, DPUs are not widely used due to their high cost, niche use cases and complex integration. Their specialized programming and limited software ecosystem make adoption difficult and they lack the general-purpose flexibility of CPUs and GPUs. However, as software support improves and costs decrease, their use may expand.

[7] https://openxla.org/
[8] https://jax.dev/
[9] https://www.nvidia.com/en-us/networking/products/data-processing-unit/

4 Design

This section outlines the project's design and setup. Figure 5 illustrates the flow of data, beginning with its capture via the SDR and ending with the processed data being written to a file. Additionally, the bottom nodes display the times and data captured from the program for testing and validation.

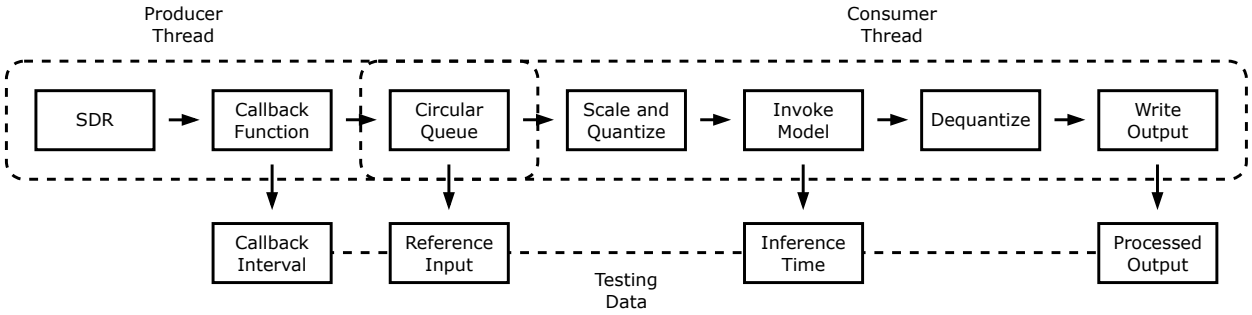


Figure 5: Data flow diagram of the program

4.1 Gathering data

The RTL-SDR USB device was used to collect samples. In the early prototyping stage, we used Python with the pyrtlsdr library[10] to control the radio. The SDR was configured with standard settings, operating at the maximum stable sample rate of 2.04 MHz. It was tuned to either 100 MHz, a busy frequency, or 80 MHz, a low-noise frequency, with the gain set to auto.
Later in the project, we transitioned to C++, which meant that we had to switch to using the official C library[11] to interact with SDR. However, this library only outputs raw byte data. To address this, we reused the logic from the Python library to shift and scale the data to a floating-point format. Additionally, we developed a wrapper around the C library to better integrate it with our C++ object-oriented design.

4.2 Holding data

The program uses two threads, one for gathering data from the SDR and the other for processing data using the Coral TPU. At times, the TPU may still be busy when the SDR collects new data. To address this waiting issue, we implemented a circular queue, also known as a ring buffer. The SDR thread places new samples into the queue, while the processing thread retrieves samples from the queue for processing.
Although the queue could handle a temporary slowdown of the TPU, depending on the queue size, it is critical that the processing thread handles data faster than the gathering thread. Otherwise, the queue will fill up, leading to data loss. Initially, we created our own simple ring buffer using locks. However, we later switched to Moodycamel's widely known lock-free ring buffer[12], which is more efficient and has been more thoroughly tested.

[10] https://github.com/pyrtlsdr/pyrtlsdr
[11] https://github.com/osmocom/rtl-sdr
[12] https://github.com/cameron314/readerwriterqueue

4.3 Preprocessing

As previously mentioned, the data from the SDR is initially converted from UINT8 to floats. However, since the Edge TPU only accepts INT8, the data is later converted back to INT8. This intermediate conversion to floats is necessary for the quantization process.
The TPU model provides input and output quantization parameters, which tell the scale and shift for the float-based model to an INT8 representation. Logically, the inputs and outputs must also be scaled to match. Although this scaling is essential for compatibility, it introduces additional overhead and precision loss.

4.4 Model Invocation

While invoking the model and processing on the Coral TPU may not appear complicated in the final code, reaching this stage was challenging. In the initial Python-based implementation, we used the PyCoral library[13] to interact with the Coral TPU. However, this library is deprecated, limiting the project to older TensorFlow versions and Python 3.9. As a result, all other dependencies had to be downgraded, compromising future compatibility.
Additionally, Python's performance became a bottleneck, prompting a transition to C++. Adopting TensorFlow Lite in C++ was a significant challenge, as newer versions of the library are only available when built from source. After numerous failed attempts using Google's own Bazel build tool[14] to compile TensorFlow Lite and Edge TPU driver libraries, we switched to a CMake[15]-based approach. Although TensorFlow itself is not compatible with CMake, its TensorFlow Lite subproject is.
Even so, the Coral driver library still needed to be built with Google's Bazel build tools. Fortunately, a maintainer still publishes prebuild driver packages[16] for newer TensorFlow versions. This solution allowed us to create a modern C++ project with the latest TensorFlow Lite versions and a simplified build process.

4.5 Outputting Data

Similar to preprocessing, the data from the Coral must be scaled and shifted back to a float. Initially, we output these float samples to stdout, but this approach proved far too slow. To improve performance, we switched to writing the data directly to a file, which significantly increased speed. For testing purposes, the program can also output additional data. For example, to validate accuracy, it writes the raw SDR samples to a file, enabling a reference FFT implementation to process the same data and compare results. The program also logs timing information, including the intervals between incoming sample batches from the SDR and the time taken to process them. Ideally, the processing time should consistently be faster than the interval between incoming batches, ensuring no samples are being dropped.

[13] https://github.com/google-coral/pycoral
[14] https://bazel.build/
[15] https://cmake.org/
[16] https://github.com/feranick/libedgetpu/releases/

5 Implementation

This section explains the process of creating our custom FFT implementation[17] within a TensorFlow model optimized to run on the Coral TPU. To begin our implementation, we first researched the Fast Fourier Transform (FFT), focusing on the Cooley-Tukey algorithm[15], which is widely used and relatively easy to implement. We started by creating a prototype using Python and Numpy, by working out the pseudocode[16] for the algorithm. We tested this against Numpy's built-in FFT function and it worked flawlessly.
The next step was to implement the same algorithm in TensorFlow, as it offers Numpy-like instructions but is the only framework supported on the Coral, with certain restrictions that would be important later. We began by using Keras, TensorFlow's easy model-building module, to create a model. This model implemented the iterative Cooley-Tukey algorithm by connecting splitting and merging butterfly layers in a tree-like structure.

Unfortunately, TensorFlow Lite, which is required for exporting models to Coral, does not support complex numbers. To work around this, we stored the real and imaginary components of the complex numbers in separate dimensions and used product multiplication to perform arithmetic on the split values. While the initial model worked, it had some issues. It generated too many nodes, making larger FFTs impossible as the compiler could not handle the model's size. Additionally, it could not run on the Coral because it used matrices with more than three dimensions, another strict requirement for the TPU.
For the third attempt, we used TensorFlow's new trace functionality, which allows to trace regular Python functions composed of supported TensorFlow operations and convert them into a model. This approach was ideal as it enabled us to recreate the earlier working pseudocode while only needing to work around the split complex numbers. However, to meet the three-dimensional limit, we had to give up batching, which became a major issue later on.

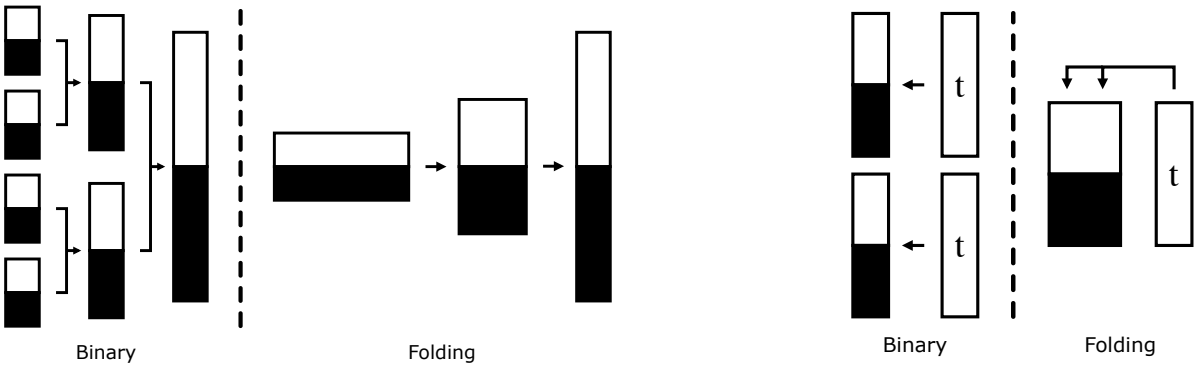
Once we started feeding the SDR's data into the Coral, another new issue emerged: The radio generated more data than the TPU could process, resulting in lost samples which is highly problematic for radio astronomy. This led to a long battle to optimize the FFT's performance, in order to keep up with the SDR's data stream. By testing with an empty model, we discovered that the bottleneck was not the FFT processing itself, but the time spent invoking the model. By processing multiple inputs together, batching allows to send as much data as possible to the Coral and reduce the frequency of loading the model. Additionally, broadcasting improves hardware utilization by enabling semi-parallel processing, which optimizes functions like applying the twiddle factors to fully leverage the TPU's specialized hardware. We returned to refactoring the TensorFlow model to enable batching while keeping to all the restrictions. After many failed attempts, we switched to using two tensors, one for the real parts and the other for the imaginary parts, freeing up a dimension and allowing us to add a batch dimension. Once we also rewrote the input and output handling in the C++ program, it finally worked.
Increasing the USB read size, meant that the read time became longer, but the Coral could process that larger batch of data in a single call, thus significantly improving performance. Timing remained tight, but writing output floats directly to a raw binary file stabilized it, preventing sample loss.

[17] https://github.com/OliVis/coral/

5.1 Folding method

The key to the latest implementation is our 'folding' technique. Instead of creating a separate node for each butterfly operation, we reshape the tensor's third dimension to match the width of the split for each iteration and stack all the butterflies in the second dimension. This allows the twiddle factors to be broadcast across all the splits simultaneously. As a result, the model only requires $lg(n)$ butterfly nodes, compared to the $2^{(n+1)} - 1$ nodes needed for a full binary tree. In addition, the twiddle factors are pre-computed, reducing the computational load for each run.
One challenging aspect was splitting the tensor into even and odd parts, as certain TensorFlow operations, such as stacking and splitting, did not compile correctly, even though they adhered to the restrictions. After some adjustments, we managed to get it working and it turned out to be a much cleaner implementation than the previous versions.

Figure 6a illustrates the difference between the standard binary tree approach, used in the initial implementation and the folding technique. For clarity, the batching dimension is left out, though it does not affect the overall concept. As shown, the folding technique requires far fewer nodes, allowing for the creation of larger FFT models with a simpler, linear graph structure. Although the input must be reshaped for folding, the output remains the same, as depicted in figure 6a. This reshaping is not an issue, as it is already necessary for the bit-reversal process[17].



(a) Graph diagrams for both models

(b) Application of twiddles in both models

Figure 6: Comparison between the standard binary method and our folding method.

Figure 6b illustrates the process of applying the roots of unity or 'twiddles' to the tensors using both the binary and folding approaches. Since the Coral TPU only supports element-wise multiplication, both methods involve the same number of calculations. However, the folding technique takes advantage of the Coral's optimization for broadcasting, somewhat performing the calculations in parallel. This approach better utilizes the Coral's architecture, requiring just a single instruction call instead of multiple nodes in the graph, which theoretically reduces execution time. Most importantly, it simplifies the graph, making it easier for the sometimes unpredictable Edge TPU compiler to handle.

The following figure 7 shows a code listing that performs the FFT on the input tensor, with the folding method implemented in the `for` loop. Additionally, note the limitations encountered at the beginning of the function:

```
def compute(self, tensor: tf.Tensor) -> tf.Tensor:
    """
    Compute the Fast Fourier Transform (FFT) of the input tensor using the
        Cooley-Tukey algorithm.

    Refer to the actual code for a more detailed definition.
    """
    # Spit tensor into the complex parts, unpacking directly doesn't work
    complex_parts = tf.split(tensor, 2, axis=2)
    real = complex_parts[REAL_PART]
    imag = complex_parts[IMAG_PART]

    # Remove the empty dimension, unstack isn't supported
    real = tf.squeeze(real, axis=2)
    imag = tf.squeeze(imag, axis=2)

    # Split the tensors into a list of complex numbers
    real_list = tf.split(real, self.fft_size, axis=1)
    imag_list = tf.split(imag, self.fft_size, axis=1)

    # Reorder the complex numbers according to the bit-reversal indices and
        concatenate them back together
    real = tf.concat([real_list[i] for i in self.bit_rev_indices], axis=1)
    imag = tf.concat([imag_list[i] for i in self.bit_rev_indices], axis=1)

    # Apply Cooley-Tukey FFT algorithm using butterfly operations
    for stage in range(self.stages):
        # Calculate butterfly size and count based on the current stage
        butterfly_size = 2 ** (stage + 1)
        butterfly_count = self.fft_size // butterfly_size

        # Reshape the tensor to prepare for butterfly operations
        real = tf.reshape(real, [self.batch_size, butterfly_count, butterfly_size])
        imag = tf.reshape(imag, [self.batch_size, butterfly_count, butterfly_size])

        # Perform butterfly operations for the current stage
        real, imag = self.butterfly(real, imag, stage)

    # Reshape the parts back to their original shape
    real = tf.reshape(real, [self.batch_size, self.fft_size])
    imag = tf.reshape(imag, [self.batch_size, self.fft_size])

    # Combine the complex parts to return the original shape, with the FFT applied
    return tf.stack([real, imag], axis=2)
```

Figure 7: Code listing for the `FFT.compute` function

The next code listing in figure 8 implements the butterfly function as defined by the Cooley-Tukey algorithm. While it maintains the same core functionality, it also incorporates batching using the previously described folding method:

```
def butterfly(self, real: tf.Tensor, imag: tf.Tensor, stage: int) -> tuple[tf.Tensor,
    tf.Tensor]:
    """
    Performs butterfly operations for the Fast Fourier Transform (FFT).

    This method applies the butterfly operation by splitting the input tensor into
        'even' and 'odd' components
    and then recombining them with applied twiddle factors for the given FFT stage.

    Refer to the actual code for a more detailed definition.
    """
    # Split the tensors into even and odd components
    real = tf.split(real, 2, axis=2) # Re, Ro
    imag = tf.split(imag, 2, axis=2) # Ie, Io

    # Store the odd components
    real_odd = real[ODD_PART] # Ro
    imag_odd = imag[ODD_PART] # Io

    # Apply twiddle factors to odds using complex multiplication
    # Twiddle factors are complex numbers that rotate the odd components in the
        complex plane
    # Broadcasting allows this operation to efficiently apply the twiddle factors to
        multiple tensors

    # Calculate real component: (Ro * Tr) - (Io * Ti)
    real[ODD_PART] = real_odd * self.twiddles[stage][REAL_PART] - \
                     imag_odd * self.twiddles[stage][IMAG_PART]

    # Calculate imaginary component: (Ro * Ti) + (Io * Tr)
    imag[ODD_PART] = real_odd * self.twiddles[stage][IMAG_PART] + \
                     imag_odd * self.twiddles[stage][REAL_PART]

    # The final butterfly operation combines the evens with the twiddled odds
    return tf.concat([real[EVEN_PART] + real[ODD_PART], real[EVEN_PART] - \
                      real[ODD_PART]], axis=2), \
           tf.concat([imag[EVEN_PART] + imag[ODD_PART], imag[EVEN_PART] - \
                      imag[ODD_PART]], axis=2)
```

Figure 8: Code listing for the `FFT.butterfly` function.

6 Experiments

In this section, we will evaluate the accuracy and performance of our finalized FFT implementation on the Coral, comparing it to CPU and GPU implementations. This comparison also includes power usage, which is a significant factor of assessing the potential of using the Coral as a hardware acceleration option. For all experiments, the SDR was configured to its maximum stable sampling rate of 2.04 MHz. Both accuracy and performance data are the average of three runs, each consisting of 10,000 iterations. While the SDR is used to gather data, the focus of these experiments is not the accuracy of the data itself but rather the capability of each FFT implementation to process the data efficiently and accurately. Importantly, the input data remains consistent across tests to ensure comparability. The spectrum samples presented below represent single samples, not averaged data.

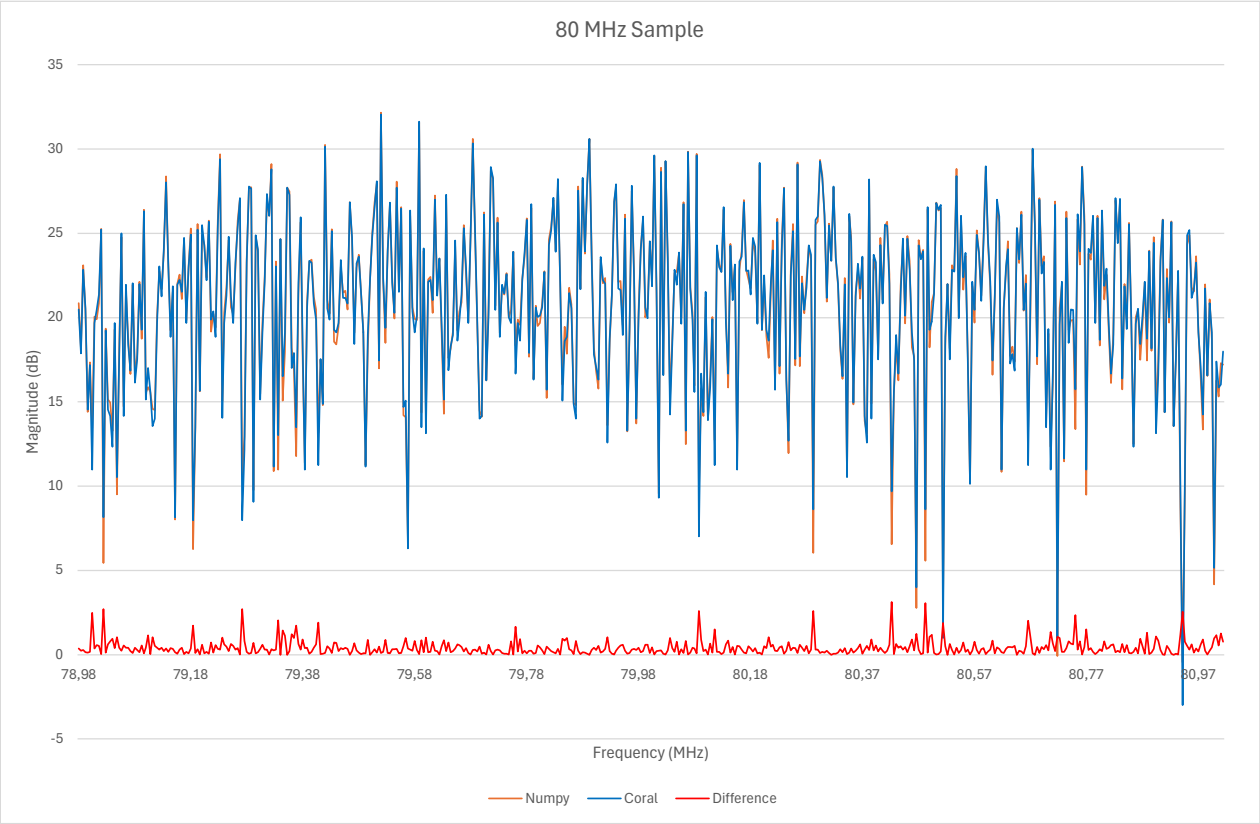


Figure 9: 80 MHz Spectrum sample computed using our Coral FFT and Numpy FFT, with the difference shown in red.

Figure 9 illustrates a sample taken from an 80 MHz capture. The blue line represents the output from the Coral, while the orange line beneath it corresponds to NumPy's FFT, serving as a reference. Additionally, the red line indicates the absolute difference between the Coral's output and the reference output.

The sample data is scaled to display values in decibels and shifted around to accurately represent the MHz frequency band for clarity. As shown in the graph, the differences between the Coral's output and the benchmark are minimal, except for slight deviations at some larger peaks. Since the 80 MHz band is not heavily utilized, the graph primarily shows a low-power, uniform signal.

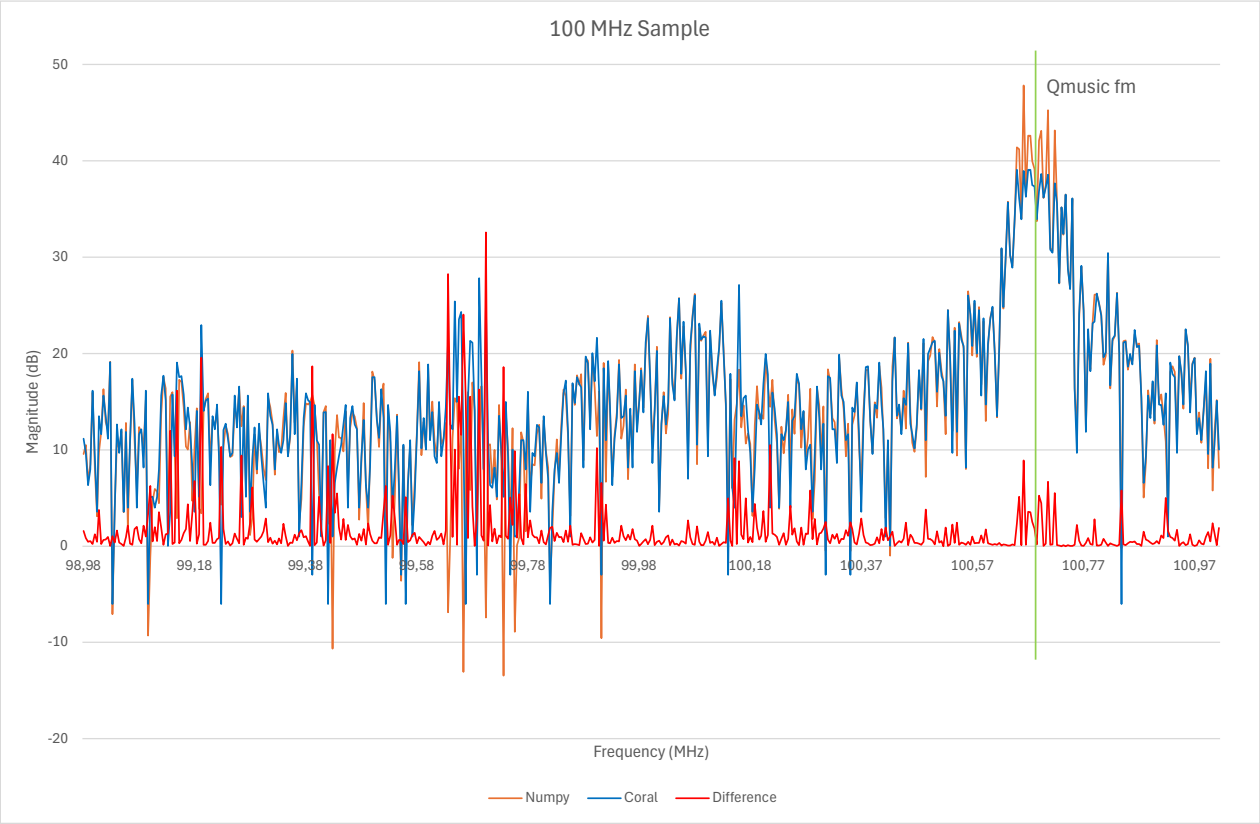


Figure 10: 100 MHz Spectrum sample computed using our Coral FFT and Numpy FFT, with the difference shown in red.

Figure 10 presents a capture at 100 MHz, a heavily used frequency range with numerous radio stations. The red difference line indicates that the error is now significantly greater compared to the previous example. While the overall shape of the spectrum is preserved, some finer details are missed by the Coral's FFT implementation.
The prominent hump on the right corresponds to the wide FM band of a radio station, while the peaks on the left represent several narrowband FM stations. The narrower peaks are significantly more often missed by the Coral, likely due to accuracy loss by quantization. Although this particular sample itself does not show it, some samples showed massive differences at these peaks, potentially pointing to byte overflow issues.
Although the accuracy of this sample is debatable, our Coral implementation is able to identify the frequencies where RFI is present, which is an important consideration.

6.1 Accuracy

While the previous two graphs represented random samples, we will now perform a more thorough analysis of the Coral's FFT output. Figure 11 illustrates the accuracy differences between the Coral's FFT and NumPy's FFT (previously depicted by the red line) in both a quiet and a busy radio frequency environment. For this test, batch size is not a factor, as it does not affect accuracy. Only the center frequency and FFT size were varied.

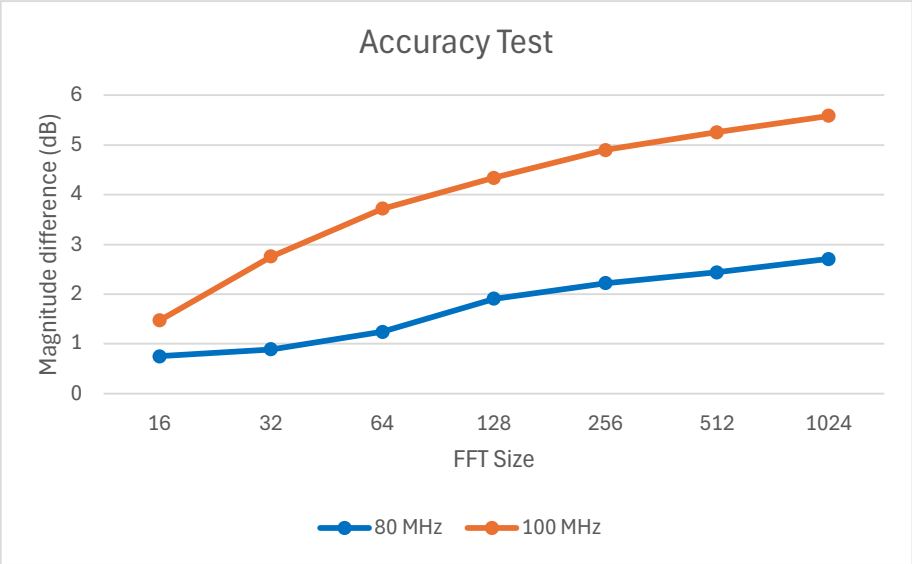


Figure 11: Accuracy comparison between our Coral FFT and Numpy FFT on different sizes and environments

As shown, the error for the quiet frequency is relatively low, whereas the error for the busy frequency is nearly twice as large across all FFT sizes. The error for an FFT size of 512 aligns reasonably well with previous graphs, which were also recorded with an FFT size of 512. Additionally, an increase in error with larger FFT sizes is expected. This is due to the exponential growth in the number of butterfly layers, where cumulative inaccuracies from each layer add up.
The accuracy loss is primarily due to quantization, where the model is converted from a floating-point format to INT8. This conversion reduces precision, leading to a gradual accumulation of errors. Although the values shown in the figure are averages and generally indicate a reasonably low error, there are significant outliers in some cases. These outliers can affect the overall usefulness of individual samples. Larger errors can arise from INT8 overflow or clipping, which occurs when values exceed the representable range of INT8. These values, which would normally fit within the floating-point range, are truncated during scaling, leading to inaccuracies.

6.2 Performance

Next, we will evaluate the computational performance. Here, we experiment with different FFT and batch sizes, measuring the time required to gather data from the SDR and the time to process it to output the final values on both the TPU and GPU. The batch size depends on the FFT size and the read size category (a combination of FFT size and batch size). Within each read size category, the same total amount of data is processed on the Coral, regardless of the FFT size. Figure 12 lists the number of batches required for each combination of FFT size and read size.
Ideally, these tests would also include a CPU-based comparison. However, the TensorFlow Lite CPU interpreter is outdated and lacks several features essential for running the FFT model, features that are available in both the TPU and GPU interpreters. It's worth noting that the GPU was unable to run any model with an FFT size larger than 128. This limitation could be from a dimension restriction in the interpreter or, less likely, an architectural constraint of the NVIDIA GTX 1080 graphics card used in the tests.

Figure 13 compares the performance of the TPU and GPU, using the same scale for both graphs. Unfortunately, as mentioned previously, the GPU test includes only half the measurements of the TPU test. Each colored line represents a read size category, with the corresponding dashed line indicating the time required to gather the samples. If this threshold is exceeded, it indicates that the TPU cannot keep up with the data captured by the SDR, leading to dropped samples.

Size	16KB	32KB	64KB
16	512	1024	2048
32	256	512	1024
64	128	256	512
128	64	128	256
256	32	64	128
512	16	32	64
1024	8	16	32

Figure 12: Number of batches per FFT size

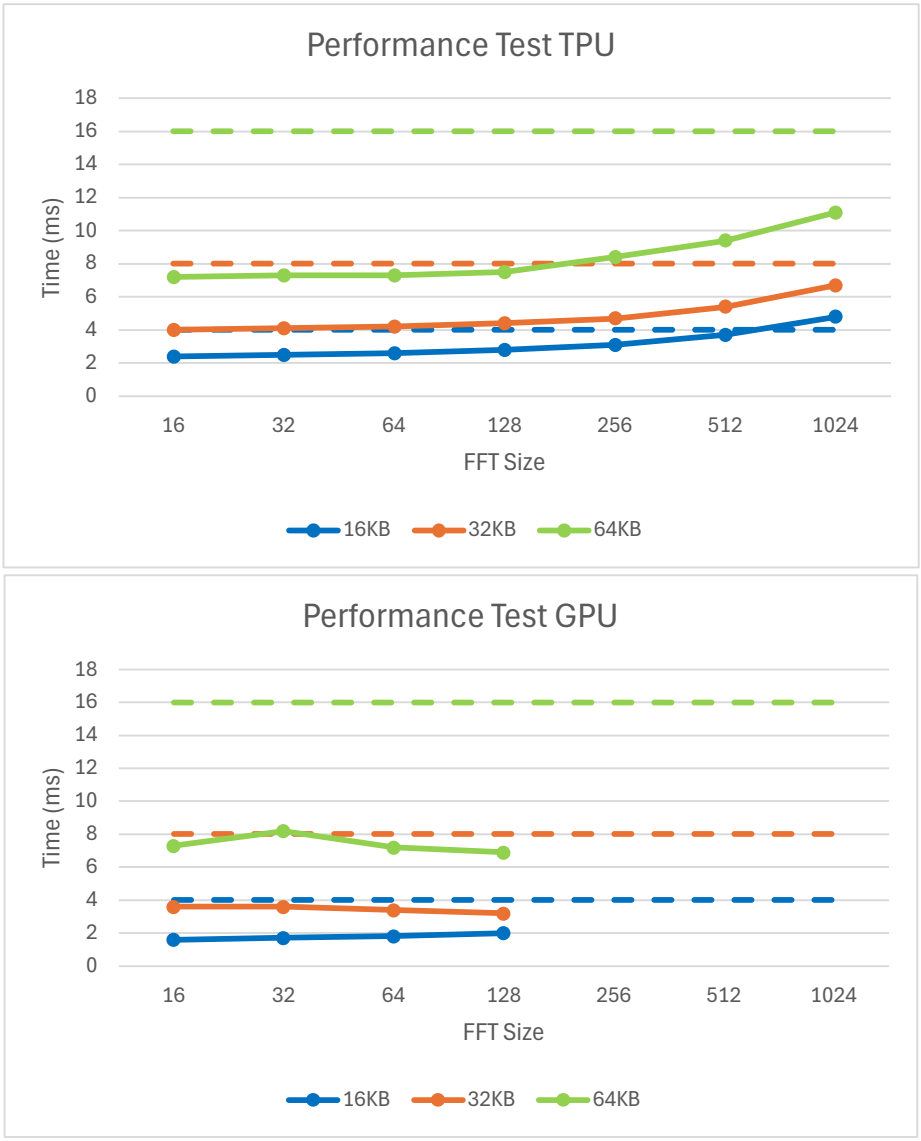


Figure 13: Comparison of processing times for various FFT sizes between the Coral Edge TPU and the NVIDIA GTX 1080 GPU, with dashed lines representing the time required to gather the samples.

Although 1024 is the largest FFT size supported by the Coral compiler, the read size can still be significantly increased. However, the usefulness of this approach is questionable, especially if the TPU can easily keep up using a smaller batch size. Smaller batches are not only faster to process but are also easier to cache or write to a file, making them potentially more efficient.
The performance of the TPU in Figure 13 follows expected trends, larger read sizes take relatively longer to process, and increasing the FFT size also results in higher output times. Notably, with larger read sizes, there is more buffer room below the gather threshold, meaning the TPU spends more time waiting for new samples. Only one data point exceeds the threshold, indicating that in nearly all cases, the TPU can keep up with the SDR as long as the batch size is configured appropriately.
In contrast, the GPU performance reveals a more unexpected pattern. While larger read sizes logically take slightly longer, the growth patterns for different read sizes vary significantly. The smaller read size slows down predictably as the FFT size increases, but surprisingly the larger read sizes show improved performance as the FFT size increases. This could be due to more efficient parallel processing or better memory utilization with larger batches. Unfortunately, due to missing data, this trend is not fully visible.
Although the GPU is slightly faster in most cases, it is far less energy-efficient compared to the Coral TPU. The performance gap, while present, is not as significant as one might expect. A more modern GPU with dedicated AI cores might perform better in these tests, but it would also come at a much higher cost, potentially 20 to 40 times the price of the Coral TPU. Additionally, other considerations such as size, heat generation, and the lack of moving parts further increase the practicality of the TPU for certain applications.

7 Discussion

While the previously shown experiments show promising results, there are concerns about how practical it is to use the Coral for this purpose. A major issue is the uncertainty about how long Google will continue to support the Coral with updates for drivers and libraries. The current documentation and software packages are already outdated, adding to these concerns.
As described in the design section, when creating a model, many unexpected challenges arise due to missing or unclear instructions, unsupported features and other limitations that are only discovered through trial and error. All these uncertainties raise concerns about the ability to build more complex models or expand the signal processing pipeline on the Coral in the future. Although the Edge TPU can store multiple models, the details about its hardware, drivers, and compiler are either unavailable or poorly documented. Furthermore, for scientific applications, we feel that it's important to use tools that are either open-source or well-documented to ensure better understanding and control of the complete system.
Despite these challenges, the Coral TPU has some clear advantages. Its performance-to-watt ratio is excellent and once the learning curve is overcome, the programming experience can be relatively straightforward. The integration with TensorFlow makes it possible to write code that is both easy to understand and powerful.

8 Conclusions

This section answers the research questions to evaluate how useful our approach is when using the Google Coral Edge TPU for signal processing tasks.

How effective are existing TPU solutions for signal processing tasks?
Large TPUs have shown good results with signal processing tasks, particularly in high-performance environments. However, for edge devices like the Google Coral, there are no available solutions optimized for signal processing. There is not even a general-computation framework available, aside from the inoperative GPTPU project. As such, despite the potential of edge TPUs in terms of energy efficiency, there are currently no viable solutions for signal processing tasks or even general-computation for these devices.

How can the workload of the SDR and TPU be distributed evenly?
This issue was significant in the early stages of the project. Increasing the number of TPU invocations kept it busier but also spent more time initializing the model rather than processing data. As a result, the TPU struggled to keep up with the SDR, and samples were being lost.
By implementing batching, the TPU could load larger chunks of data at once, increasing the relative time spent processing instead of loading. This allowed it to process all the data more efficiently, preventing sample loss. As shown in the experiments, the time required to process the data is very close to the time needed to collect it, meaning both systems operate near their maximum capacity together.
The SDR can generate a large amount of data quickly while the TPU processes it intensely. If batching is still not enough to prevent data loss, the SDR's sampling rate can be reduced to decrease the amount of incoming data.

Is the Coral Edge TPU a suitable accelerator for signal processing tasks?
This question can be answered using our FFT implementation on the Coral. The Coral's computational power is much lower than a GPU, but it performs very well in terms of power efficiency. However, its effectiveness depends on your specific needs. On the downside, developing for the Coral can be challenging due to limited documentation and support. Given these factors, it's only a good idea to use the TPU if you really need its energy-efficient performance and are willing to deal with development challenges and the uncertainty of future support.

9 Future work

While the FFT model showcases a solid foundation for the use of edge hardware in these tasks, there is significant room for further research to make it more suitable for real-world applications. In particular, additional signal processing steps are needed for tasks like RFI detection. For example, models for beamforming or polyphase filters could be developed, these models could then be loaded onto the Coral, enabling the data to pass between invocations in a pipeline. Ideally, this would allow the model and signal data to remain on the TPU throughout the entire pipeline, improving efficiency. The Coral also supports splitting large models across multiple TPUs, which could be leveraged to build more complex pipelines.

Another potential application of signal processing models on the Coral is the integration with GNURadio. This would enable easy pipeline creation while leveraging the TPU's computational power.

Beyond signal processing, a highly valuable project would be the development of a general computing framework for the Edge TPU. Such a framework could use TensorFlow Lite functions internally that adhere to the TPU's limitations. This would significantly simplify development and could attract more interest in the Coral for various applications.

References

[1] A. R. Thompson, J. M. Moran, and G. W. Swenson, *Radio frequency interference*. Cham Springer, 1 2017.
https://doi.org/10.1007/978-3-319-44431-4_16.

[2] E. O. Brigham and R. E. Morrow, *The fast Fourier transform*, vol. 4. IEEE, 12 ed., 12 1967.
https://ieeexplore.ieee.org/document/5217220.

[3] M. P. Van Haarlem, M. W. Wise, A. W. Gunst, G. Heald, J. P. McKean, J. W. T. Hessels, A. G. De Bruyn, R. Nijboer, J. Swinbank, R. Fallows, M. Brentjens, A. Nelles, R. Beck, H. Falcke, R. Fender, J. Hörandel, L. V. E. Koopmans, G. Mann, G. Miley, H. Röttgering, B. W. Stappers, R. A. M. J. Wijers, Zaroubi, and et al., "LOFAR: The LOw-Frequency ARray," *Astronomy and Astrophysics*, vol. 556, p. A2, 5 2013.
https://www.aanda.org/articles/aa/abs/2013/08/aa20873-12/aa20873-12.html.

[4] R. van Nieuwpoort, "Towards exascale real-time RFI mitigation," *2016 Radio Frequency Interference (RFI)*, pp. 69–74, 10 2016.
https://ieeexplore.ieee.org/document/7833534.

[5] Y. Sun and A. M. Kist, "Deep Learning on Edge TPUs," *arXiv.org*, 8 2021.
https://arxiv.org/abs/2108.13732.

[6] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: a framework and analysis," *IEEE Xplore*, pp. 306–317, 2 2003.
https://ieeexplore.ieee.org/document/1176259.

[7] M. Mishra, A. Potnis, P. Dwivedy, and S. K. Meena, "Software defined radio based receivers using RTL — SDR: A review," *IEEE Xplore*, pp. 62–65, 10 2017.
https://ieeexplore.ieee.org/document/8378125.

[8] K.-C. Hsu and H.-W. Tseng, "Gptpu: Accelerating applications using edge tensor processing units," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 6 2021.
https://doi.org/10.1145/3458817.3476177.

[9] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed., 2010.
https://dl.acm.org/doi/10.5555/1891996.

[10] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
https://ieeexplore.ieee.org/document/1386650.

[11] J. Kim, S. Hyeon, and S. Choi, "Implementation of an sdr system using graphics processing unit," *IEEE Communications Magazine*, vol. 48, no. 3, pp. 156–162, 2010.
https://ieeexplore.ieee.org/document/5434388.

[12] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," 2019.
https://arxiv.org/abs/1907.10701.

[13] T. Lu, Y.-F. Chen, B. Hechtman, T. Wang, and J. Anderson, "Large-scale discrete fourier transform on tpus," *IEEE Access*, vol. 9, pp. 93422–93432, 2021.
https://ieeexplore.ieee.org/document/9465154.

[14] J. Hu, P. A. Bernstein, J. Li, and Q. Zhang, "Dpdpu: Data processing with dpus," 2024.
https://arxiv.org/abs/2407.13658.

[15] J. W. Cooley and J. W Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, pp. 297–301, 4 1956.
https://www.jstor.org/stable/2003354.

[16] T. H Cormen, C. E Leiserson, R. L Rivest, and C. Stein, *Introduction to Algorithms, third edition*. MIT Press, 3 ed., 7 2009.
https://books.google.nl/books?id=i-bUBQAAQBAJ.

[17] J. Rius and R. De Porrata-Doria, "New fft bit-reversal algorithm," *IEEE Transactions on Signal Processing*, vol. 43, no. 4, pp. 991–994, 1995.
https://ieeexplore.ieee.org/document/376852.