Exploration of the AMD Ryzen NPU for Real-time Signal Processing

Real-time Imaging of LOFAR Station Data

Master's Thesis Report by
J.A. Fortanet Capetillo

TU Delft

Exploration of the AMD Ryzen NPU for Real-time Signal Processing

Real-time Imaging of LOFAR Station Data

by

J.A. Fortanet Capetillo

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday June 30, 2025 at 14:30.

Student number:	4893093	
Project duration:	November 1, 2024 – June 30, 2025	
Thesis committee:	Dr. H. P. Hofstee	TU Delft & IBM, supervisor
	Prof. Dr. Ir. A. J. van der Veen,	TU Delft
	Ir. S. C. van der Vlugt,	ASTRON
	Dr. Ir. Z. Al-Ars,	Trinilytics BV
	Dr. M. D. Ruiz Noguera,	AMD

An electronic version of this thesis is available at
https://github.com/antonio-fc/lofty_Ryzen_NPU.

TU Delft

Acknowledgments

I would like to express my gratitude to everyone who supported me throughout my thesis.

First, I am grateful to ASTRON for providing me with the opportunity to conduct my thesis work within their organization. I would like to thank Steven van der Vlugt, Corné Lukken, Mattia Mancini, Cees Bassa, and Jan David for their continuous guidance, technical support, and for sharing their expertise during my time at the company. Their knowledge of astronomy and computer systems was essential for the shaping and realization of the thesis project. Also, I am grateful to everyone else in the company who welcomed me and assisted me during my visits to the ASTRON offices.

I am very thankful to my university supervisors, Peter Hoftee, Zaid Al-Ars, and Arjan van Genderen, for their continuous support and feedback throughout the development of the thesis. I also want to thank Christiaan Boerkamp in particular for his help in understanding the TINA framework and for his assistance in the development process.

I also want to thank AMD and their MLIR-AIE development team for assisting me in the project by providing me with computer platforms for development, and for answering my questions throughout the thesis. In particular, I would like to thank Mario Ruiz Noguera for taking the time to listen to my questions and providing me with the necessary information to advance the project's development.

This work received funding from the European Union through the RADIOBLOCKS (101093934) project and the Dutch Research Council (NWO) through the DAS-6 (621.018.201) project. Also, I would to extend my gratitude to the AMD Hackster Pervasive AI Developer Contest for providing the price funds necessary to purchase the Phoenix system.

Furthermore, I would like to thank my friends from CESE (and Ir. friends). I will always remember the movie nights and going to Yosshi with them, as well as the necessary support and inspiration they have given me to get to this point. I would also like to thank my friends from the IP, who have given me moral support and many laughs during my time in Delft. I extend my appreciation to all the friends I made in Delft for making my time here so much more enjoyable.

Last but not least, I want to thank my family for their unconditional love and support, not only during this thesis but throughout my life. This was a difficult process, but thanks to them, I found the strength to continue until the end. None of this would have been possible without them.

J.A. Fortanet Capetillo
Delft, June 2025

Summary

The rise of Artificial Intelligence (AI) applications has driven the development of accelerators offering high performance and energy efficiency for AI-powered applications. Among these is the Ryzen Neural Processing Unit (NPU), which is integrated into AMD's Ryzen AI processors and has been demonstrated to have significant potential for accelerating AI workloads. This thesis explores the feasibility of repurposing this AI-focused hardware for Digital Signal Processing (DSP) applications, particularly in the field of radio astronomy. This is done by using the All-Sky Imaging Algorithm employed by ASTRON in the LOFAR system as a case study.

The current CPU implementation of the All-Sky Imaging Algorithm cannot meet the real-time processing requirements imposed by the 10 Hz data generation rate from the LOFAR telescope stations. This creates a performance bottleneck in the diagnostics image generation pipeline of the stations. To address this, four implementations of the algorithm were developed. Three of the implementations were developed using MLIR-AIE, while the fourth implementation was developed using TINA. Both of them are toolchains used to implement applications to run in the Ryzen NPU architecture. The solutions explore various parallelization and pipelining strategies to maximize the performance of the implementations while maintaining correctness and minimizing power consumption.

The experimental results show up to 77.4 times speedup over the CPU baseline and 2.84 times over the GPU implementation, with 3 out of the 4 implementations achieving the 10 Hz real-time image generation rate. All the solutions produced accurate results, with minor variations in precision due to the difference in implementation data types. Unfortunately, there is a lack of power consumption results for the NPU implementations to compare with the CPU and GPU implementations. Nevertheless, the findings suggest that the Ryzen NPU is a viable platform for accelerating DSP applications and provides guidance for future implementations.

This thesis contributes an NPU-accelerated implementation of the All-Sky Imaging Algorithm, adds a layer to the TINA toolchain, as well as an example using the Ryzen NPU platform, and provides a proof of concept for future DSP applications on the Ryzen NPU. Future work includes enabling power measurements, expanding algorithm support for better accuracy and flexibility, and accelerating other LOFAR station applications by leveraging the Ryzen NPU.

Contents

1 Introduction

In the last decade, there has been a major shift in the computing industry. With the decline of Moore's Law [1] and the rise of Dark Silicon [2, 3], to keep up with the surge of larger and denser algorithms, it was necessary to look for new ways to increase the performance of computationally intensive applications. This led to the emergence of Domain-Specific Accelerators (DSA) [4]. Rather than using conventional CPU architectures, which are capable of executing almost any kind of application, DSAs use alternative architectures better suited for more specific fields such as graphics, robotics, simulation, and many others [5]. DSAs work similarly to GPUs and FPGAs, in that they are typically used as coprocessors alongside the CPU to execute specific parts of an application that benefit from the specialized architecture. However, unlike GPUs and FPGAs, DSAs are designed to have an even narrower application range, further increasing performance.

At the same time, there has been an explosion in the number of Artificial Intelligence (AI) and Machine Learning (ML) applications being developed and released to the public. Their capabilities have surpassed expectations of what was once thought possible with these kinds of models. From the now well-known Large Language Models (LLM) [6] to image and video generation Diffusion models [7], ML is changing the way we consume and create digital content. DSAs for AI applications can offer an alternative that can dramatically increase performance while keeping power consumption low [8].

1.1. The Rise of AI Accelerators

Nowadays, AI accelerators have become a common strategy to maximize the performance of AI/ML applications. An AI accelerator is a device whose hardware architecture is specialized for running AI/ML applications to minimize execution time and resource utilization. Compared to the standardized designs of CPUs and GPUs, AI accelerators comprise a diverse set of hardware architectures [2, 9]. Nevertheless, a common strategy is the use of "data-flow" architectures [10] that physically resemble the structure of the applications they are designed to execute. Other common features in these systems are high-parallelism capabilities and high-memory bandwidth for fast data transfer [9].

AI accelerator devices have since been integrated into heterogeneous systems, such as the Ryzen 7040 and 8000G models, which combine CPU, GPU, and NPU into one desktop computer (moving forward referred to as Ryzen AI). The term NPU (Neural Processing Unit) refers to an AI accelerator designed to mimic the processing function of the human brain [11]. The Ryzen 7040 is the first x86 processor with an integrated NPU, and it has shown great promise for low-power execution of ML applications with up to 33 times higher performance per Watt [12, 13] compared to CPU implementations. The new models using the Ryzen AI processors have only continued increasing their capabilities, which leaves the need to test their potential uses.

1.2. Just AI?

DSAs are typically designed to maximize performance for a narrow range of applications. As a result, deviating from this may result in a worse performance than simply executing on a general-purpose CPU. However, this can leave potential applications unexplored compared to the wide use of general-purpose devices. For instance, Ryzen AI processors, which are intended for ML applications, have the

potential to be used in the field of Digital Signal Processing (DSP) because of their high parallelism and high throughput requirements, which are also common characteristics of ML applications. By mapping the matrix operations and streaming structure of DSP applications [14] to the vector processors and dataflow hardware of the Ryzen NPU, performance improvements can be achieved in terms of execution time and power efficiency. This approach has the potential to bring DSP implementations closer to the efficiency levels seen in ML applications by using the same hardware.

1.3. Using AI Accelerators for Astronomy

In astronomy, the use of telescopes has been essential for observing celestial bodies and phenomena since the time of Galileo, allowing astronomers to observe celestial bodies that are too distant for the naked eye. In modern times, observing only in the visible light spectrum is not enough to get the full picture of what is out there. Therefore, telescopes capable of observing lower and higher frequencies of the electromagnetic spectrum have become just as important. Because of this approach to astronomical research, organizations have appeared to focus on certain frequency ranges. The focus on observing low-frequency electromagnetic waves is also referred to as radio astronomy.

The Radio Astronomy Institute of the Netherlands, also known as ASTRON, is a Dutch research institute that utilizes cutting-edge technology and the collaboration of various scientific fields to enable radio astronomy discoveries. LOFAR (LOw Frequency ARray) is the main telescope system at ASTRON, which is comprised of a series of stations spread across the Netherlands and Europe [15]. Each station has two types of omnidirectional antennas, targeting higher and lower ranges of radio frequencies, respectively. The signals from the antennas can be converted to digitized in the time domain, and if needed, converted to the frequency domain. The frequency data can be used to analyze the observations in the form of spectrographs [16], while the time data can be computationally combined to generate images. In contrast to single-dish telescope systems, the images generated with the data from LOFAR have higher quality. This happens because the distance between the antennas emulates the total diameter of a single dish antenna, thus increasing the image definition (more details of this concept are explained in Section 2). Because of this, all the stations can cooperate in making discoveries in radio astronomy that were previously impossible.

1.4. Current System and Challenges

The data processing in the LOFAR station consists of several stages that transform the incoming radio waves into usable data for scientific research. The first step is the Analog-to-Digital Converter (ADC), which digitizes the radio waves. After digitization, the signals are filtered to remove noise from environmental and human-made sources. The next steps are correlation and beamforming. Correlation is done to obtain the signal from the source without noise, and beamforming allows the system to focus on a certain portion of the sky, emulating the use of a dish while using omnidirectional antennas. The processes of ADC, correlating, and beamforming are done online. The output from this data path is then sent to COBALT, a computer cluster in Groningen dedicated to the intensive processing of the LOFAR station data [17].

The signals from the antennas of each station are also sent to a processor on-site for smaller tasks, such as image generation. These are relatively lower-quality images that are used mainly for calibration and quality control. These are also useful for diagnostic purposes by informing operators of hardware malfunctions and environmental noise, which could potentially reduce the quality of results down the line. However, the current implementation used to generate diagnostics sky images cannot keep up with the rate of data generation of the LOFAR station antennas, causing a bottleneck in the system.

Each of the LOFAR stations has a Linux machine as its main configuration, management, and diagnostics infrastructure. The diagnostic imaging application runs on the CPU of the system, with an image generation rate of 1.8 Hz. This latency does not allow the system to create images from the incoming signals in real-time, which are generated at a rate of 10 Hz. At the same time, the available power supply and environmental conditions of the station cabinet impose tight constraints on the processing power that can be used in the station cabinet.

1.5. Problem Statement and Research Question

The necessity of implementing a real-time imaging application with the available resources in the LOFAR system presents a very interesting opportunity to test the capabilities of the Ryzen AI system for signal processing applications in the field of radio astronomy. The sequential imaging algorithm shows a high potential for parallelism as well as the ability to be computed in independent sub-operations. These characteristics further justify using Ryzen AI hardware, which is designed to accelerate applications with such characteristics.

The development of the imaging application with the Ryzen NPU, in turn, results in a solution for the issues affecting the LOFAR on-site image processing. The objective of this implementation is to obtain a real-time application by increasing performance while maintaining correctness and matching or reducing power consumption. The new application serves as an example and proof of concept for the development of similar applications, whether for radio astronomy or an entirely different application of signal processing methods. Ultimately, the objective is to show the potential of running DSP applications in the Ryzen AI architecture for the sake of exploration. Also, the result can help to open the door to new uses for this specialized hardware that is becoming progressively more common in commercial systems.

With all this in mind, the research question for this thesis becomes: **To what extent can the All-Sky Imaging Algorithm be accelerated using the AMD Ryzen AI NPU, and how does this impact performance and resource utilization compared to baseline CPU and GPU implementations?**

1.6. Methodology

In order to have a thorough evaluation of the Ryzen AI system, the imaging algorithm will be implemented using different programming frameworks, as well as different heuristics that vary in performance and resource use. From these different implementations, a comparison will be made, and the best version will be selected based on its performance, resource use, and fitness in the field. Furthermore, the NPU implementations will also be compared to CPU and GPU-based implementations, working as baselines for the performance results.

The programming frameworks used for these implementations are: MLIR-AIE [18] and TINA [13] (further explained in Section 3). Through profiling and benchmarking, this thesis aims to gain insight into the potential performance gain of the application at hand with the addition of the NPU to its pipeline. Furthermore, it aims to explain the achieved performance of the different implementations in terms of performance (execution time vs work) by using the Roofline Model and other similar methods, as well as power consumption. Other than comparing the implementations regarding performance and resource use, the thesis also aims to briefly discuss the user experience when working with the frameworks used on each of them, comparing ease of use and versatility to the obtained results.

1.7. Thesis Outline

In Chapter 2, the relevant background information about radio astronomy and hardware accelerators is presented to give context to the reader regardless of background. Chapter 3 explains the tools and technologies used for this implementation and their main features. In Chapter 4, the different solutions are explained in terms of their heuristics, as well as their potential performance increase. Chapter 5 discusses more technical details of the solutions presented in Chapter 4 to justify the decision-making in some of the design features. Chapter 6 presents the results for all the implementations, and how these compare to the theoretical predictions and the baselines. In Chapter 7, the results are discussed, the limitations of the project are presented, and how this can be translated into future work, and finally, the conclusion summarizes the project's findings.

2

Background

2.1. Radio Astronomy

2.1.1. A Brief History

The concept of radio astronomy originated in the 1930s, when an engineer from Bell Laboratories, Karl G. Jansky, was investigating the persistent noise present in transatlantic communications at the time [19]. After months of work, he concluded that the noise was not coming from Earth, but rather from the sky. Another observation was that this noise was shifting across the sky. With the help of astronomers, he was able to conclude that he was measuring radio waves coming from the center of the Milky Way [19]. This, in turn, revolutionized the field of astronomy, creating Radio Astronomy.

Since Janksy, the discoveries made by radio astronomers have completely changed humanity's perception of our galaxy and the universe as a whole. The most notable contribution from radio astronomy is the now famous 'Cosmic Radio Background', as seen in Figure 2.1, which further cements the Big Bang theory as the start of our universe. Additionally, certain astronomical bodies, such as galaxies, quasars, neutron stars, and hydrogen clouds, are only observable with radio telescopes. With these observations, scientists can determine the chemical composition and behaviors of these objects that were previously invisible. Since radio waves can penetrate through most dust and gas in outer space, these can reach us further than any other wavebands, allowing us to make observations further away than ever before, and even low-energy radiation, like from cold celestial bodies, can be detected within this range [20, 21].

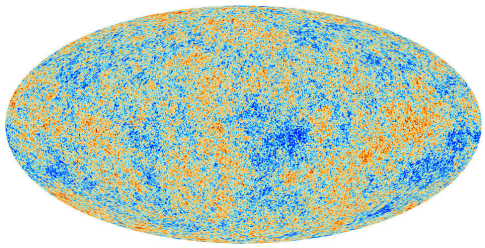


Figure 2.1: 2-dimensional projection of the 'Cosmic Radio Background', obtained from [22]

2.1.2. Basic Theory

The range for the observation of the wave frequency of radio astronomy is usually located between 10 MHz and 1 THz in the electromagnetic spectrum, as seen in Figure 2.2. This is quite logarithmically broad compared to other types of electromagnetic radiation; thus, it is quite common to focus on narrower frequency bands when building telescopes and doing research[20, 21].

An important advantage of radio astronomy compared to the study of other frequency bands is the presence of atmospheric windows. These are the frequency ranges in the electromagnetic spectrum that can pass through the Earth's atmosphere, which causes problems for detection in most of the electromagnetic spectrum. These windows are significant because only radio waves within these windows,

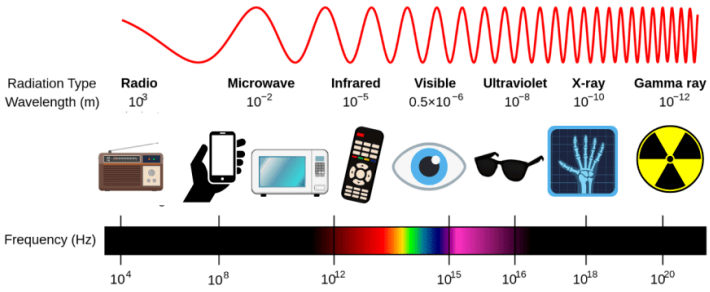


Figure 2.2: Electromagnetic Spectrum, obtained from [23]

along with parts of the optical and infrared (IR) spectrum, can be detected from the Earth's surface, as seen in Figure 2.3. This property of radio waves enables the development of more sophisticated ground-based telescopes, unlike those for X-rays, ultraviolet (UV), and gamma rays, which require space-based observatories due to their inability to penetrate the atmosphere [20, 21].

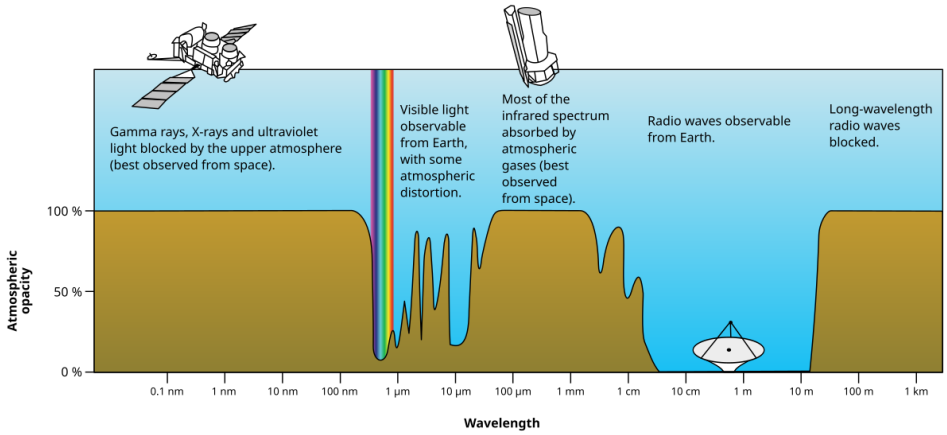


Figure 2.3: Atmospheric Windows of Electromagnetic Radiation, obtained from NASA

2.1.3. Telescopes

When observing distant objects, such as celestial bodies, two factors limit the ability to detect them. The first, and the most well-known, is that as an object is further away, it appears smaller because of the phenomenon known as perspective. The second is that as objects are further away, fewer photons bouncing off them can reach the observer, thus making them appear dimmer. When it comes to astronomy, these are the main limiting factors that determine how far observations can be made.

Telescopes are devices with a long history of use in astronomy. Since the 17th century, telescopes have become essential in astronomical research by allowing the user to amplify as well as magnify the images seen through them. These capabilities solve the two issues previously mentioned, thus expanding the range of observations for astronomers. The intensity is increased by taking in light through a larger area, called *aperture*, and making the light rays converge into a smaller area, called *focal point*, allowing them to be taken in by the eye of the observer. The light gathering amplifies the intensity of the incoming light and provides the observer with a sharper picture, and depending on the design of the telescope, also provides magnification, thus resulting in a larger picture of the object being observed. The early telescope designs, and many optical telescopes today, use mirrors and glass lenses to bend light rays to the focal point of the telescope, also known as Newtonian telescopes, which are named after their inventor [24, 25].

Traditionally, to make observations in astronomy, a researcher only required a single telescope or dish to make observations. However, singular telescope configurations become less effective as the target becomes farther away. This is because telescopes have a given angular resolution that limits the distance of the objects they can observe. Angular resolution refers to the smallest angle between two

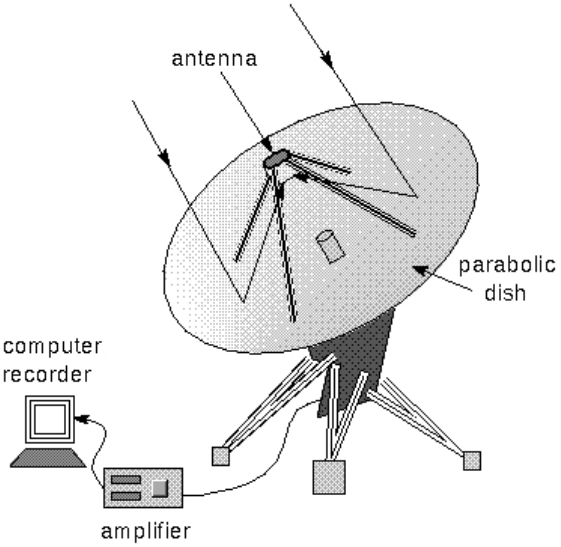


Figure 2.4: Radio Antenna Dish Amplification, obtained from [26]

points that can be seen as clearly distinct [27]. This is due to diffraction, which is the bending of light waves on the aperture walls, which happens when the wavelength is longer than the aperture diameter. This phenomenon sets a limit for traditional telescopes because the longer the distance to the target, the larger the telescope must become to be able to detect it. However, it gets to a point where the build becomes infeasible, so a new method is needed to break this barrier.

The limited angular resolution of single-telescope setups spurred the development of telescope arrays for the generation of higher-quality astronomical images. This technique is called *Interferometry*. However, to combine the signals coming from several telescopes, they need to be computed using several different techniques depending on the telescope setup and desired result.

Radio Interferometry

Interferometry is a technique that switches the large parabolic antenna for a set of (relatively) smaller antennas to receive signals from the same astronomical source, and then combines the signals to extract spatial information. In doing so, the antenna set simulates a telescope with a diameter equal to the maximum separation between any two antennas in the set. The set of relative distances between any two antennas in an antenna array is called *baseline*. [28]

At its core, radio interferometry obtains the correlated component of two signals to isolate the signal coming from a common source in the sky, and suppress the uncorrelated noise coming from unrelated sources; these are referred to as *correlated visibilities* [28]. Correlated visibilities are a set of complex numbers obtained from each pair of antennas that indicate the intensity of the correlated signal (real component) and the phase given the spatial difference of the antennas (imaginary component). The intensity and, therefore, observed distance of the interferometry process is given by the number of sampled antennas. The correlated visibilities can then be used to reconstruct an image of the sky.

2.1.4. The All-Sky Imaging Algorithm

The All-Sky Imaging Algorithm is used to transform the signals received by a set of antennas into an image, and in so doing, it projects the sky as a 3-dimensional sphere to a 2-dimensional plane in a process called *aperture synthesis*. As seen in Equation 2.1, the algorithm is a variation of the more famous Inverse Discrete Fourier Transform (IDFT) algorithm, which calculates each of the recreated signal samples based on the frequency components. The algorithm was adapted from a formula in [28] by switching the baselines from 2 to 3 dimensions. In the case of the All-Sky Imaging algorithm, the formula calculates the brightness intensity for each of the image pixels, which represent a directed observation in the sky. The computational complexity is $O(n)$ as a function of the number of pixels of the image, and $O(n^2)$ as a function of the number of sampled antennas.

$$Img[l_{\text{ix}}, m_{\text{ix}}] = \Re\left\{\frac{1}{M \cdot N}\sum_{i=0}^{M}\sum_{k=0}^{N} V_{i,k} \cdot \exp\left(-2\pi j \frac{f}{c}\left(u_{i,k} \cdot \mathsf{l}_{l_{\text{ix}},m_{\text{ix}}} + v_{i,k} \cdot \mathsf{m}_{l_{\text{ix}},m_{\text{ix}}} + w_{i,k} \cdot \mathsf{n}_{l_{\text{ix}},m_{\text{ix}}}\right)\right)\right\} \quad (2.1)$$

The purpose of the All-Sky Imaging is to reconstruct the image from the measured visibilities V, which are obtained from the correlated visibilities. They are represented as a 3D tensor, which can be subdivided into two matrices that represent the real and imaginary components of the complex measurements in the frequency domain. These are obtained for a given frequency f.

In the IDFT algorithm, the frequency series is assumed to be evenly spaced in the frequency and time domains, which in the All-Sky imaging algorithm is not the case. This is where the baselines $\{u, v, w\}$ come in. These represent the relative distance for each pair of N antennas, and it depends on the antenna field layout. These are then scaled with the cosine directions $\{l, m, n\}$ to project them to specific directions in the sky corresponding to each of the pixels of the image. The cosine directions are given by the unit vector pointing to the source point of the to-be pixel. Thus, these have to satisfy $l^2 + m^2 + n^2 = 1$, where l is the direction along the East-West axis, m is the direction along the North-South axis, and n is the direction along the horizon.

Once the baselines are added for a given pixel calculation, these need to be converted from spatial physical units to spatial units in terms of the wavelength for f. To achieve this, the scaled and added baselines are multiplied by the scalar $-2\pi jf/c$. Now the phase can be obtained by applying the exp function, like in the original IDFT, representing the sinusoidal nature of the electromagnetic waves. Finally, this gets multiplied by the visibilities, aggregated, and normalized to obtain the pixel value. Since the purpose of the calculation is to generate an image, the imaginary component of the output can be discarded.

Two aspects of the algorithm make it a good candidate for a parallel implementation. The first is that the pixels can be computed independently, allowing the computation to occur in parallel. The second is that the operations in the inner loop are all element-wise operations (except for the final mean operation). Therefore, the elements in the inner loop are also fully independent from each other. This would allow for the parallel calculation of each data point index, followed by the mean at the end.

Listing 2.1 shows a simple Python implementation for the All-Sky Algorithm, which serves as a good starting point for any attempts at accelerating the application. The inputs are assumed to be already formatted as required by the algorithm and are therefore not taken into account when measuring the program's performance. The input size of this algorithm is given by three factors:

1. Number of sampled antennas
2. Dimensions of the output image
3. Computation data type

Equation 2.2 shows how to calculate the input size for this basic implementation. The number of antennas determines the size of the visibilities. Due to the correlation process, the visibilities size is given by $NumOfAntenna^2$, and this is multiplied by two because of the real and imaginary components. Similarly, the baselines size is given by $NumOfAntenna^2$ times 3 for $\{u, v, w\}$. The cosine directions size is equal to the number of pixels times 3, for $\{l, m, n\}$. Finally, it adds 1 for the sampling frequency scalar.

$$InputBytes = (1 + 5 * NumOfAntenna^2 + npix\_l * npix\_m * 3) * DataTypeBytes \quad (2.2)$$

The input data for the algorithm are typically obtained from stored data for the visibilities, sampling frequency, and baselines. On the other hand, the cosine directions can be generated for each implementation since they are not dependent on the input data, but only on the output image size. The way to obtain the cosine direction in Listing 2.2 is given by the dimensions of the image $npix\_l$ and $npix\_m$. The calculation of n results in a large portion of the values being imaginary when $l^2 + m^2 > 1$, which occurs for $(1 - \pi/4) * 100\% = 21.46\%$ of them because of the nature of the projection. When $m^2 + l^2 > 1$, then the value of n^2 is negative given the unit vector constraint, meaning the projection is trying to go below the horizon, thus giving a meaningless output for a 2d projection of the sky.

Listing 2.1: All-Sky Imaging Algorithm implementation in Numpy

```
def all_sky_image(vis, u, v, w, freq, l, m, n, npix_l, npix_m):
   img = np.zeros((npix_l, npix_m))
   for l_ix in range(npix_l):
      for m_ix in range(npix_m):
         img[npix_l - l_ix - 1, npix_m - m_ix - 1] = np.mean(
                  vis
                  * np.exp(
                     -2j * np.pi * freq
                     * (
                        u * l[l_ix, m_ix]
                        + v * m[l_ix, m_ix]
                        + w * n[l_ix, m_ix]
                     ) / SPEED_OF_LIGHT
                  )
               )
   return np.real(img)
```

Listing 2.2: Code for generating the cosine directions l, m, n.

```
l, m = np.meshgrid(np.linspace(-1, 1, npix_l), np.linspace(1, -1, npix_m))
n = np.sqrt(1 - l**2 - m**2) - 1
```

2.1.5. ASTRON

ASTRON was established in the 1980s as the successor to the SRZM (Foundation Radio Waves from the Sun and the Milky Way), which was in charge of the development of ASTRON's first system, called the Westerbork Synthesis Radio Telescope (WSRT), to explore the newly developed radio frequency technology. Then, in 2010, LOFAR was opened and has since then been the largest radio telescope system in the world, with stations in many European countries.

In 2023, the LOFAR European Research Infrastructure Consortium was formed to coordinate the exploration and improvement of the LOFAR system. [29, 30] Since its foundation, ASTRON has been a major contributor to the field of radio astronomy. For instance, LOFAR was used for the discovery of radio aurorae, which is used for the discovery of exoplanets [31]. Additionally, it has served as inspirational leadership to define and design the Square Kilometer Telescope Array (SKA) in Australia.[32]

LOFAR

LOFAR is a telescope is a radio telescope designed to operate at the lowest frequencies that can be observed from Earth [33]. LOFAR is a multipurpose sensor network that is capable of working in the frequency range from 10-240 MHz. The system consists of an array of dipole antenna stations distributed throughout the Netherlands and 9 other European countries. There is a total of 54 stations, of which 38 are in the Netherlands and the rest in the countries of Germany (6 stations), Poland (3 stations), the UK, France, Ireland, Latvia, Sweden, Bulgaria and Italy (See map in Figure 2.5). All these stations work in collaboration under the International LOFAR Telescope (ILT) foundation, founded in 2010 [29].

In the Netherlands, the core of the system places 24 of the stations within a 2 km diameter, located approximately 30 kilometers away from ASTRON's offices in Dwingeloo. The stations in this location are called "core" stations. Each of the stations is comprised of 48 High-Band Antennas (HBA) and 96 Low-Band Antennas (LBA) as shown in Figure 2.6. The rest of the stations in the Netherlands are more dispersed across the north and northeast of the country, extended across a radius of 90 km; these are referred to as "remote" stations. The remote stations have the same number of LBAs and HBAs. Finally, the stations outside of the Netherlands are called "international" stations and are equipped with 96 HBAs and 96 LBAs. [33, 15]

For each station, the antennas are connected to a hardware cabinet. This is a small metallic shed filled with hardware used to collect, pre-process, and process the signals coming from the antennas through coaxial cables. The results coming from the cabinet are then sent to the offices of the respective

Figure 2.5: Map of the LOFAR stations across Europe, obtained from [33]

organization for further processing and analysis [33]. For the LOFAR core, these signals are sent to the Dwingeloo offices or COBALT. COBALT (COrrelator and Beamformer Application for the LOFAR Telescope) is a GPU-based software correlator and beamformer used by ASTRON for the creation of higher-quality images used for scientific analysis [17].

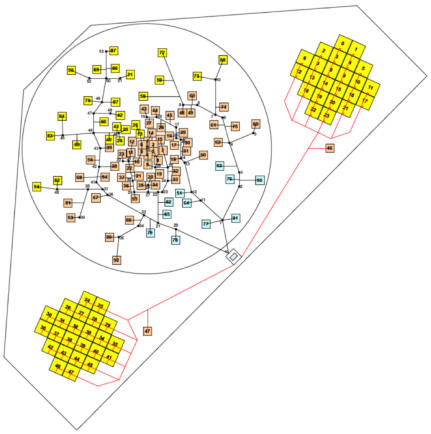
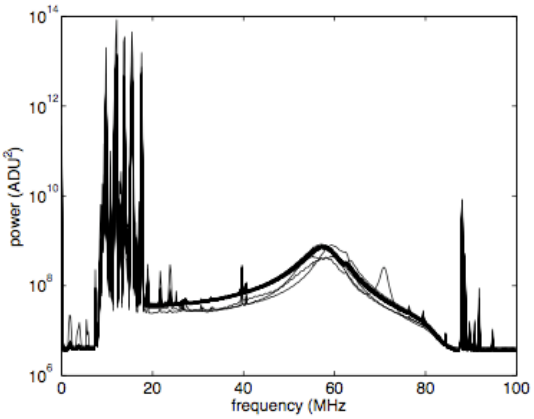


Figure 2.6: LOFAR station layout with LBAs and HBAs, obtained from [33]

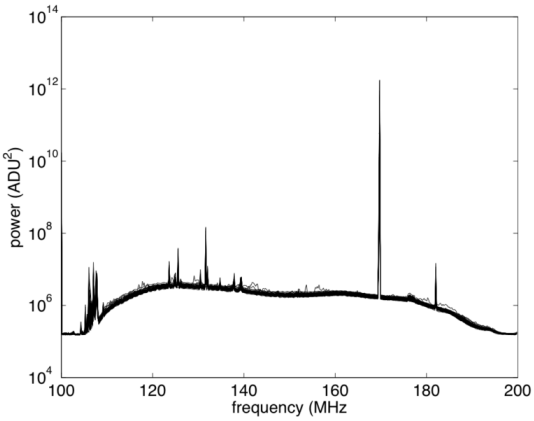
The LBAs are designed to operate between the cutoff of the radio atmospheric window at 10 MHz to the onset of the commercial FM radio at 90 MHz. The LBAs are built as a simple dual linear polarization droop dipoles above a conducting ground plane, with the wires at an angle of 45 degrees with the ground. The LBAs suffer from a relatively high degree of Radio-Frequency Interference (RFI) at the ends of their frequency range as seen in Figure 2.7(a). In the lower frequencies, this is due to ionospheric reflections of sub-horizon RFI, and in the higher frequencies, this is due to FM radio. To minimize interference, the LBAs employ a filter to suppress it [33].

On the other hand, the HBAs are designed for detecting signals between 120 MHz and 240 MHz. At this range, the noise coming from the sky no longer dominates the total system noise like with the LBAs. Therefore, these have a different design. The HBAs are also dual dipoles, but these are arranged in tiles of 4x4 antennas. Each HBA tile is equipped with an analog radio frequency beamformer, which combines the signals from the 16 antennas in phase for a given direction. The HBAs also experience some amount of RFI, but it is not as prevalent as with the LBAs and is more dispersed across their frequency range, as seen in Figure 2.7(b).

The station cabinet signal path can be divided into three parts: the Receiver Units (RCU), the Science Data Processor (SDP), and the Local Control Unit (LCU). The LCU is a Linux machine that monitors and controls the local hardware station. The LCU is described in Section 3.1, and this is where the All-Sky Imaging runs. The other three components function as a pre-processing pipeline for the signals coming from the antennas, as seen in Figure 2.8. The diagram describes the signal path for



((a)) Frequency response for LBA dipoles of LOFAR, obtained from [33]



((b)) Frequency response for HBA tiles of LOFAR, obtained from [33]

Figure 2.7: Frequency response for LBA and HBA of LOFAR

the newly upgraded LOFAR 2.0 system, which is currently being installed.

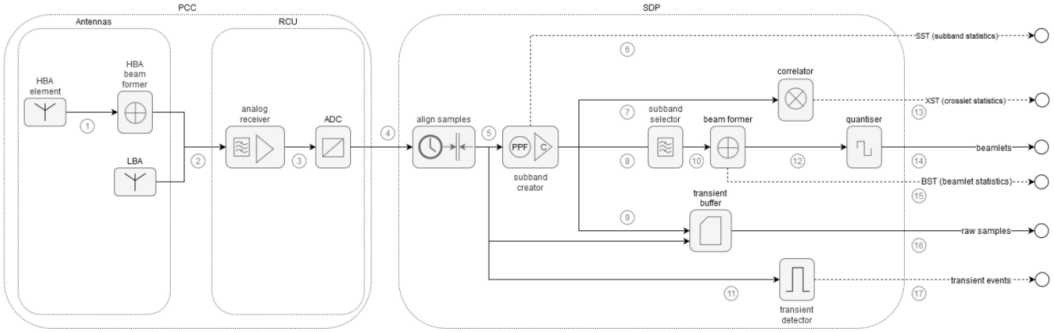


Figure 2.8: LOFAR 2.0 signal path, flowing from the station to the COBALT supercomputer center, provided by ASTRON

The RCUs are the components that gather the signals coming from each of the antennas of the station and digitize them for processing. The antennas are connected to the RCU through three single-polarization inputs. There is an RCU for each HBA tile or LBA pole. The incoming signal is passed through a filter, which is a 10 and 30 MHz high-pass filter for the LBAs, and 110-190, 170-230, and 210-250 MHz filters for the HBAs. For the LBAs, the filters are to suppress the strong Earth-bound short-wave radio signals. For the HBAs, the filters are used to select an appropriate Nyquist zone for a certain clock and observing frequency. For the HBAs, this is also where the beamforming of the tiles is done. The filtered (and beamformed) signals are then digitized by a 12-bit analog-to-digital converter (ADC) at a sampling frequency of 160 or 200 MHz, depending on the bandwidth requirements [33]. There are 4 RCUs per station, which are then connected to a single SDP.

Once the signals are digitized, they are then passed to the SDP for the creation of a variety of data products. The SDP largely consists of FPGAs, which provide effective acceleration for signal processing and allow for easy reconfiguration if necessary. The SDP provides data for both the LCU and also for the COBALT data processing center. As seen in Figure 2.6, the signals go through several data paths for the creation of the outputs. The most relevant for this thesis is the creation of the crosslet statistics (XST), which are the correlated visibilities used in the All-Sky Imaging algorithm. In order to get the XSTs, the digitized signals must first be separated by subband frequency, which corresponds to the sampling frequency of the visibilities. The subbands are limited to frequency buckets within the range of the given antenna type. Then, the subband signals are sent to the correlator, and finally, the output results in the XST correlated visibilities.

Ultimately, the signals coming out of the SDP are sent to COBALT at a rate of approximately 3 to 9 GB/s (depending on the station) per station through a wide area network [15]. After going through COBALT at the LOFAR central processor, the results go to CEP (Calibration and Imaging) and finally to

off-site data archives. The data generated at the end of this pipeline is what scientists use for analysis.

2.2. Hardware Acceleration and AI

Hardware acceleration refers to the use of specialized hardware devices to perform specific types of tasks more efficiently than running on general-purpose CPUs. This concept uses the principle that any data transformation that can be executed in software on a generic CPU can also be done using dedicated hardware. Hardware accelerators are commonly not a replacement for the CPU, but rather a supplement that takes care of the ”accelerated workloads” that are better suited for the hardware-accelerated platform, while the rest of the operation can still run in the CPU [2].

In order to improve computing performance, one can invest in better hardware, better software, or a combination of the two. This comes with trade-offs in latency, throughput, and energy consumption. Software solutions typically offer better flexibility, faster development, and lower up-front costs. On the other hand, hardware solutions can provide significant speed improvements, lower power consumption and latency, increased parallelism, and more efficient resource use. The downsides of hardware-based solutions are their limited upgradability and a much longer development time. Nevertheless, in spite of the advantages of software solutions, these are ultimately limited by their hardware platform, so in order to maximize performance for a given application, developers must consider the running platform for their implementations and how switching to a better-tailored device may improve performance.

Hardware acceleration is used in a wide range of fields to improve performance by offloading tasks to dedicated hardware. These include Computer graphics, Digital Signal Processing, Genome Sequence Alignment, Sound Processing, Computer Networking, Cryptography, and AI, among many others. The field greatly influences the hardware requirements, given the variety of algorithms and performance requirements. However, the most common strategy for hardware acceleration is to exploit parallelism. Parallelism is the capacity for an application to run in multiple processing units at the same time while maintaining correctness. This is reflected in the most common hardware acceleration platforms, which are Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs). These two device types are composed of sub-processing units capable of running in parallel to greatly improve application performance and also give users great flexibility for development in any field.

As mentioned previously, higher flexibility usually means lower performance potential, thus creating the need for Domain-Specific Architectures (DSA). These are devices built specifically to tackle the needs of a specific field. This usually means having hardware components that are specially built for very common operations used in the algorithms of said field. An example of this kind of device is [34], which is an accelerator designed for genome sequence alignment. An even more extreme version of this concept is the Application Specific Integrated Circuits (ASIC), which are built to tackle a specific application, such as a chip running for a voice recorder or an encryption chip [35].

2.2.1. AI Accelerators

AI accelerators are one of the most common types of hardware accelerators nowadays due to the large presence of AI-powered applications such as Large-Language Models (LLM), image and video generators, and speech recognition, just to name a few. AI accelerators are a specialized subset of hardware accelerators designed to handle AI workloads, in particular, large-scale neural network inference and training [9]. Unlike general-purpose processors, AI accelerators are optimized for operations common in ML models, such as matrix multiplication and convolutions, as well as supporting highly parallel architectures and high data throughput. These devices range from more general-purpose GPUs to devices like TPUs, FPGAs, and ASICs; each with a different balance of flexibility, performance, and resource efficiency.

Despite their utility, AI accelerators introduce a number of challenges when it comes to their development and utilization. One of the main driving factors when designing AI accelerators, other than GPUs, is power consumption. Especially for devices used for very large-scale models, power consumption has become a concern for environmental reasons due to a large amount of energy consumption, putting a strain on power grids and accelerating the effects of Climate Change [36]. Additionally, in edge computing applications, power consumption is also important because of the thermal and battery constraints, causing overheating and damage to the surrounding hardware. According to [37], by 2030 the electricity demand from data centers worldwide is set to double, consuming more than 945 TWh, of which AI is projected to be the most significant contributor to this increase. The new object of

many AI accelerator implementations is to reduce the impact of these applications while maintaining the performance of increasingly powerful AI models.

A major concern with the use of AI accelerators is the lack of standardization across platforms. Unlike CPUs, which have benefited from decades of well-established and mature programming environments and standardization, AI accelerators often require specialized compilers, libraries, and low-level APIs to utilize their capabilities effectively. For instance, for developing an application using an FPGA, the developer needs to be familiar with Hardware Description Languages (HDL), while to implement the same application but with a GPU, the development process must be done using CUDA, OpenCL, or similar. There exist attempts at unifying a programming model such as MLIR [38] and PyTorch [39], but the development landscape is still largely fragmented.

Despite its shortcomings, AI accelerators have achieved great success in enabling AI applications by drastically reducing execution time and resource use. Because of the variation in architectural designs, the achieved acceleration can vary widely. According to [9], data center chips can achieve performance of up to 1000 Tera-operations per second (TOPS). This increase in performance has also been observed to increase power consumption at a similar rate [9]. This trend has created the need for AI accelerators that make power efficiency a priority while maintaining the performance of the AI tools that have become ubiquitous today.

2.2.2. Dataflow Architecture

A very common strategy for designing AI accelerators is called dataflow architecture [9]. Rather than executing instructions sequentially under a single program counter, dataflow-driven devices execute instructions based on data availability. In turn, this enables a high degree of parallelism and energy-efficient processing [9].

In the dataflow execution model, a program is represented by a directed graph. The nodes of the graphs are instructions, where each node may perform different operations [40]. The nodes may be limited to a single operation, but they may also compute multiple instructions per node. The directed arcs are the data dependencies between them, behaving in a First-In-First-Out manner (FIFO)[40]. Thus, the execution of a program is given by the flow of the input data from the input nodes to the leaf nodes, giving the output of the program. In dataflow-driven devices, the nodes represent the individual processing units, and the arcs represent the connections between them. Each of the processing units may have its own memory space, but it is also possible to have shared memory between them. A comparison between a dataflow vs traditional execution models can be seen in Figure 2.9, where each of the figures represents the same application with its respective execution models.

((a)) Example program following the traditional execution model.

((b)) Example program following the dataflow execution model.

Figure 2.9: Implementations of an example program using the traditional and dataflow execution models.

2.2.3. Systolic Arrays

The systolic array is an on-chip multiprocessor architecture comprised of many identical processing cells, all connected to their nearest neighbor [41, 42]. This is a subset of dataflow architectures where the computations are propagated through the array with a throughput proportional to the bandwidth [41]. This type of architecture has become common for running implementations in the fields of ML and DSP.

3

Technologies Used

This chapter describes all the tools used to implement the All-Sky Imaging application, both hardware and software. These are described by their relevant characteristics and how they are to be used in the project. Additionally, a clear justification is provided for their use in place of alternative options.

3.1. Hardware

The target system for this thesis is the AMD Ryzen 7 8700G. This system is part of the Ryzen 8000 series, which uses the Zen4 processor architecture and has the code name ”Phoenix”, which is going to be the shorthand name for the system model moving forward in the report. The Zen4 processor has the following specifications:

- The CPU has 8 physical cores with 16 threads in total.
- The CPU base clock is 4.2GHz, but it can go up to 5.1Hz
- It has integrated GPU (iGPU) capabilities with the AMD Radeon 780M
- The chip includes an AI accelerator (NPU) with the brand name AMD Ryzen AI
- The NPU architecture is the XDNA, which arranges a set of AI Engines (AIE-ML) designed to maximize performance for AI applications
- The reported performance of the NPU is said to be up to 16 TOPS [43]

The Phoenix system was chosen over other AI-accelerated platforms because it is similar to the LCU in the station cabinets of the LOFAR 2.0 stations. Currently, the NPU in these Phoenix systems is not being utilized. By using this underused hardware for an important and computationally intensive task, we not only make better use of existing resources but also reduce the load on the CPU, allowing it to handle additional tasks more effectively.

To simplify the development process, ASTRON acquired a Phoenix system to remain in the office, which will be referred to as ”Phoenix1”. By connecting the Phoenix1 to the DAS-6 computer cluster at ASTRON [44], it can be accessed through an SSH connection, which further facilitates the development and testing of new software applications designed for the LOFAR stations. The Phoenix1 is installed with Ubuntu 22.04, as per specifications to use the AMD tooling.

3.1.1. XDNA Architecture

The XDNA architecture is a DSA designed to maximize the efficiency of AI workloads. This architecture is used for the NPU device in the Phoenix and was the iteration for this line of devices. It has been used in other notable devices such as the Versal Adaptive SoC, which combines the XDNA AI Engines with programmable logic [45]. In 2024, a newer version of the architecture was released, called XDNA2, which was used to develop more powerful Ryzen AI devices, such as the Ryzen AI 300 series [46].

The XDNA architecture is comprised of a 2D systolic array [41] of hardware units called ”tiles”. The tiles are connected in a feedforward fashion to create a spatial dataflow architecture [10], which means

that data streaming through the tiles drives execution. These tiles are arranged into 5 columns of 6 tiles and can function independently of each other, enabling a high degree of parallelism. The tiles are connected through interconnect modules present in each of the tiles, and they manage the mechanisms to allow for inter-tile communication.

The first communication mechanism is the AXI4-Stream Interconnect network, which uses a dedicated data movement unit called AXI4-Stream Interconnect. The movements can be done to/from the neighboring tiles in all 4 directions of the array: North, South, East, and West. The number of connections to tiles depends on the direction, where the movements from North to South are capable of 6 connections, and the rest are capable of 4. The network allows for the movement in and out of the tile's memory with a data width of 32 bits per connection.

The second communication network is the Memory-mapped AXI4 Interconnect network. This works thanks to the memory-mapped AXI4 interconnect, which enables memory and register access to internal tile resources. This works similarly to the AXI4-Stream Interconnect, where the tiles from all 4 directions can be accessed, but the final destination must be a direct neighbor. This connection allows for loading speeds of 512 bits/cycle and store speeds of 256 bits/cycle. Thus, the transfer speeds between tiles are significantly faster between neighboring tiles than non-neighboring tiles.

There are three types of tiles in the array. The first is the Shim Tile (ST), whose function is to move data to and from the array and main memory. The second is the Memory Tile (MT), which is used to distribute and join data streams, an operation not possible with the other tiles. And finally, the Compute Tile (CT) is where the computation happens. As seen in Figure 3.1, the array has 5 columns, and for each column, the tiles are numbered in 1 Shim Tile (purple tiles), 1 Memory Tile (green tiles), and 4 Compute Tiles (blue tiles) per column, except for the first column which does not have an ST.

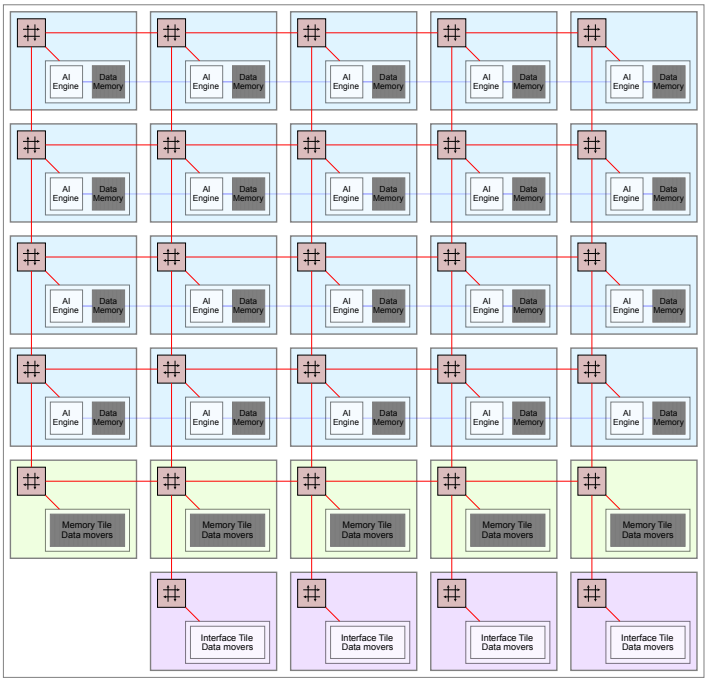


Figure 3.1: Ryzen AI Array, obtained from [43]

Compute Tiles

The Compute Tiles (also called Compute Cores in some of the documentation) are Very Long Instruction Word (VLIW) processors that support single-instruction multiple data (SIMD) for both fixed and floating-point data types. The tile features the memory and stream interconnect units, the AI Engine, where the processing happens, and the Data Memory.

The processing unit of the Compute Tile is called AI Engine. There are two types of AI Engines mentioned in the AMD documentation, which have different capabilities that must be presented to avoid confusion moving forward. The first is the regular AI Engine (AIE), these are exclusive to the Versal and

is designed for Signal Processing and ML applications. The second is the AIE-ML, which is available in multiple platforms, including the Phoenix. As the name suggests, this version is better tailored to run AI and ML applications. Because of these differences in purpose, there are some features relevant to this report:

- The AIE-ML supports more diverse datatypes commonly used in ML applications, such as int4 and bfloat16. The bfloat16 datatype is the same as float32, but it has 16 fewer bits for the mantissa.
- The regular AIE hardware supports non-linear operations such as trigonometric, square root, and inverse operations, while the AIE-ML does not.
- AIE-ML does not have native support for float32, it can be emulated through bfloat16

The VLIW functionality of the AIE-ML is managed by the Instruction Fetch and Decode Unit (IFDU). This unit issues the current program counter (PC) value as an address to access the program memory, which responds with a 128-bit wide instruction. After fetching, the instruction is decoded, and the corresponding control signals are distributed to the Compute Tile's functional units. The program memory has a capacity of 16 KB, enabling storage of up to 1024 instructions, each 128 bits in width. However, the instruction size may vary from 16 to 128 bits and supports multiple instruction formats to reduce program memory size. The full instruction size of 128 bits corresponds to 2 loads and 1 store from data memory, 1 scalar operation, 1 vector operation, and 1 register move from either processing unit.

The AIE-ML features two processing units, one for scalar operations (Scalar Unit) and the other for vector operations (Vector Unit). The Scalar Unit is a 32-bit scalar RISC processor with a 32x32 multiplier and a 32-bit ALU. It does not have a floating-point unit but supports it through emulation. The second is the Vector Unit, which has 2 subunits to handle Fixed-Point and Floating-Point vectors, respectively, and both share the Vector Register Unit.

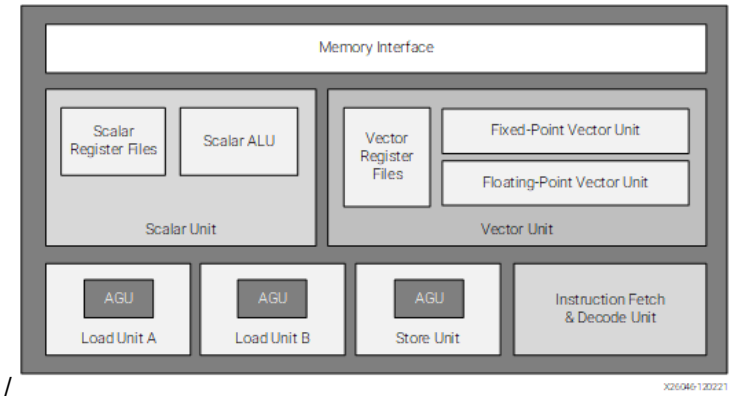


Figure 3.2: High-level structure of the AIE-ML, obtained from [47]

The Data Memory Unit composition can be seen in Figure 3.3 has the capacity for 64 KB, organized as 8 memory banks, and each is a 512-word x 128-bit single-port memory. From the programmer's perspective, these are 4 memory banks of 16 KB, each built from 2 physical memory banks that act as ping-pong buffers to facilitate the fast switching between reading and writing data in and out of the tile. The memory bank is an arbitration round-robin mechanism to prevent starving requesters of memory access. One request can be handled per cycle, so if multiple requests are made in a single cycle, one is accepted and the rest are stalled until the next cycle.

Shim Tiles (Interface Tiles)

Shim Tiles are responsible for moving data between the NPU device and main memory. They read external memory and put it into a stream interface to be sent to a Memory or Compute tile. There is also the option to broadcast the same stream to multiple tiles simultaneously. Each shim tile can process 2 input and 2 output streams. This means that each tile can support an incoming stream to the NPU and an outgoing stream to main memory simultaneously. It also supports compression when the data is sparse (when the data has many zeros).

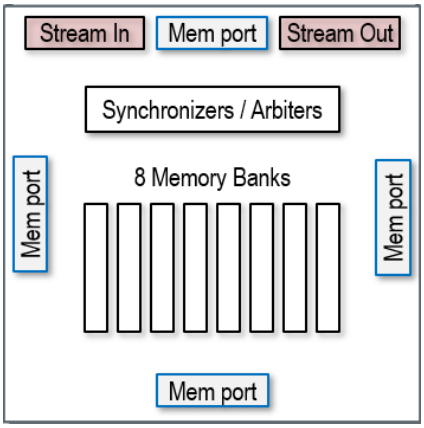


Figure 3.3: Data Memory Unit of the Compute Tile, obtained from [43]

Memory Tiles

The Memory Tile was introduced with the AIE-ML architecture to significantly increase the on-chip memory inside the array. It has a similar structure to the Compute Tile but without the AIE-ML processor. The MT has a total Data Memory of 512 KB with a high bandwidth of 30 GB/s parallel read and write. The data memory is split over 16 memory bands of 32 KB each [47].

The Memory Tile has 6 stream-to-memory and 6 memory-to-stream data movers. These are used to move data from anywhere in the array. The main functionality of these data movers is to have the capacity to distribute an incoming data stream into up to 6 separate streams or to join up to 6 data streams into one. This allows the imbalance of 8 input/output streams and 20 CTs to be resolved by virtually having up to 6 streams per ST.

Tracing

The NPU array also has tracing capabilities via two trace streams coming from each tile. One stream comes from the AIE-ML, and the other comes from the memory module. The two streams are connected to the tile interconnect unit. There is a trace unit in each AIE-ML and memory module in a CT, and one for each ST.

The trace output comes in packets of 8x32 bits, including one word of header and seven words of data. The packets are sent to the main memory of the device through the same data transfer interface as the application data. The trace records events on the tiles with the operation type and the time stamp in cycles since the start of the application.

XDNA driver

The XDNA driver enables the operating system to recognize and use the NPU device [48]. Without the driver, the AI features fall back on the CPU. The current version of the driver for Linux does not support the use of the first column of the NPU array [49]. This column is the furthest on the left, which does not have a Shim Tile. Therefore, the implementations that utilize the Ryzen AI NPU are limited to using 4 columns. This leaves availability for 4 STs, 4 MTs, and 16 CTs.

3.2. MLIR-AIE

MLIR (Multi-Level Intermediate Representation) is an open-source compiler infrastructure developed by the LLVM [50] project. Its development aims to address the challenges of modern hardware programming. MLIR introduces the concept of multiple levels of abstraction, allowing for more effective optimization of code compatible with various domains and hardware targets. Most notably, it has uses in the development of AI applications through integrations with TensorFlow and the support for compilation for various hardware platforms, such as GPUs, TPUs, FPGAs, and quantum processors [38].

Due to the emerging nature of Ryzen AI systems, application development for systems integrated with an AMD NPU is still in its early stages, with only a few available methods. MLIR-AIE is a specialized extension of MLIR designed to support the development of applications using the AMD AI Engine. It

provides a series of dialects that facilitate the mapping of high-level descriptions into the distributed AIE-enabled hardware [51].

One of the programming toolchains used for this project is called Interface Representation for hands-ON (IRON), which is used for close-to-metal programming of the AIE array developed by AMD. IRON is an open-source toolkit based on Python language bindings around the MLIR-AIE dialect [18]. IRON has two different programming styles, the ”basic” and the ”placed” styles. The difference is that the basic style defines tile behavior implicitly, where the compiler assigns the AIE tiles. On the other hand, the placed style describes the design at the tile level of granularity, where components are explicitly placed on AIE tiles using coordinates. Moving forward, the description will use the placed style, given the higher level of control over the design.

An IRON project consists of 3 main components: the NPU code, the host (CPU) code, and the C++ kernel(s). In general terms, the NPU code is what determines the data movements between the NPU tiles and what programs are executed in each of the tiles. The host code prepares the input, calls the NPU application, and receives the output of the NPU application. And finally, the C++ kernels are the programs running in the individual AI Engines of the NPU. The project is linked through the use of a Makefile for convenience, but this is not a requirement. A simplified version of the application building process can be seen in Figure 3.4.

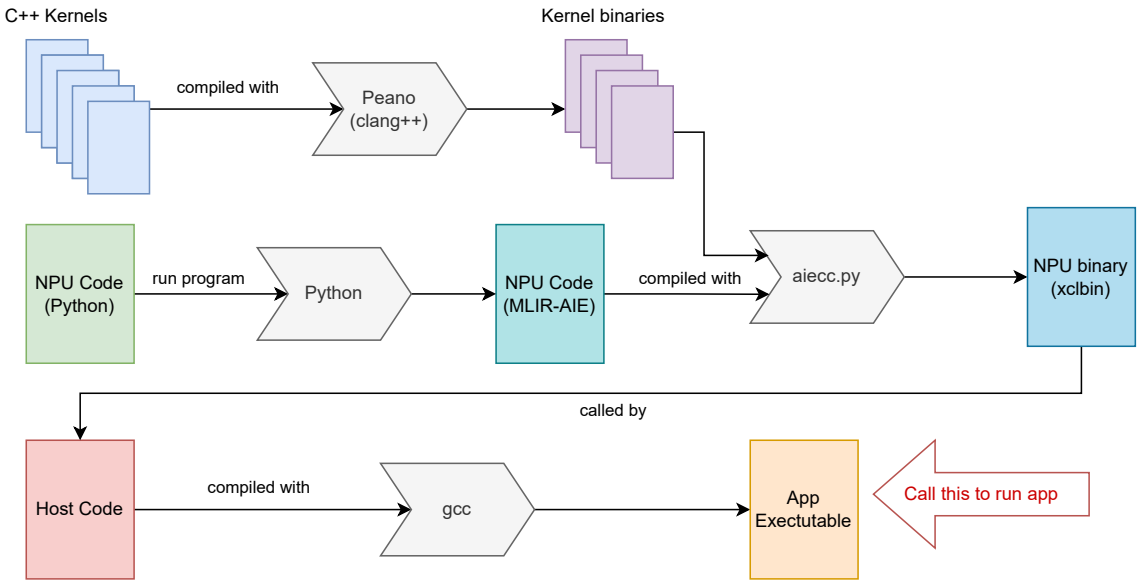


Figure 3.4: IRON application building process from its starting components.

3.2.1. Host Code

The host side is a C++ or Python program where the input data is generated to be sent to the NPU. The program is enabled to communicate with the NPU through the *xrt* library. An XRT kernel *xrt::kernel* object is used to run the NPU application and takes the input data from XRT buffer objects *xrt::bo*. The process of generating the input data can be done through the use of vectors or arrays of the type of choice. The contents are then copied to the buffer objects. After populating the buffer objects, these are synced with the device to reserve the memory space for copying the data.

For getting the output NPU application execution, the host also needs to define a buffer object with the size of the output. Once this is completed, the output data can be copied to the container of choice. Listing 3.1 shows a simplified version of the host code.

```
#include "xrt"

auto xclbin = xrt::xclbin(xclbin); // Load the xclbin
auto xkernels = xclbin.get_kernels(); // Load the kernel (the NPU code)
auto kernel = xrt::kernel(hw_context, kernelName); // Get a kernel handle

auto bo_input = xrt::bo(device, IN_SIZE, XRT_BO_FLAGS_HOST_ONLY, group_num_In);
// Generate the input and copy to the buffer object bo_input
auto bo_output = xrt::bo(device, OUT_SIZE, XRT_BO_FLAGS_HOST_ONLY, group_num_Out);

bo_input.sync(XCL_BO_SYNC_BO_TO_DEVICE);
xrt::run run = kernel(opcode, bo_instr, instr_v.size(), bo_input, bo_output);
bo_output.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

// Use the output from buffer object bo_output
```

Listing 3.1: Skeleton for host side code using C++

3.2.2. NPU Code

The NPU code is written in the IRON Python bindings, representing the data movements and computations done in the NPU. The Python bindings get converted to the MLIR-AIE dialect simply by running the Python script. Then, along with the C++ kernels, it gets compiled to a Xilinx Compiled Binary *xclbin* file, which is a file format usually used for device configuration containing hardware configuration data, kernel metadata, linkage information, and other data needed for the device at hand [52]. The code is compiled with the script located in *aiecc.py* of the MLIR-AIE library. The code for the NPU is divided into 4 main parts: the preamble, memory interfacing, data movement connections, and core definitions.

The Preamble

This part refers to the declaration of the device, the kernels, and the relevant values that will be used throughout the code, whether they are constant or obtained through command-line arguments. The kernel definitions must be tied to external binary files generated using the Peano compiler back-end.

This section is where the program defines the amount of NPU resources used in the application. The compiler allows for the optional utilization of array columns, by choosing to use 1 up to 5 columns. This is done by passing a dialect constant to the *device* function, which wraps the entire device definition for internal connections and tile behavior.

For the Phoenix NPU architecture, there are a total of 5 device constants. The first is *AIEDevice.npu1*, which signals the use of all 5 columns. The other four have the naming convention of *AIEDevice.npu1_Ncol*, where *N* can be 1, 2, 3, or 4, which signals said number of columns used.

Memory Interfacing

This determines how the input data is moved by the Shim tiles to and from main memory. This determines the number of Shim Tiles used, the size of the data movement chunks, and to which tiles are subsequently sent. This is also where the Tracing function is enabled and routed to the Main Memory.

This part of the program is configured through a function called *sequence*. The arguments for this function are mapped to the buffer objects given to the kernel object in the host code (not to be confused with the CT kernels). Then these are linked as input streams to the Shim Tiles, thus allowing for 4 input streams. In case the tracing is used, another buffer object needs to be defined on the host side. A simple example of the sequence block with tracing enabled can be seen in Listing 3.2.

Listing 3.2: Object FIFO definition between two tiles

```
trace_utils.configure_packet_tracing_flow(tiles_to_trace, trace_shim_tile)
# To/from AIE-array data movement
@runtime_sequence(full_input_ty, full_input_ty, full_input_ty, full_input_ty)
def sequence(input, output):
   trace_utils.configure_packet_tracing_aie2(tiles_to_trace, trace_shim_tile,
       ddr_id=4, trace_size, trace_offset)
   npu_dma_memcpy_nd(metadata=of_in, bd_id=1, mem=input, size) # input
   npu_dma_memcpy_nd(metadata=of_out, bd_id=0, mem=output, sizes) # output
   dma_wait(of_out)
   trace_utils.gen_trace_done_aie2(trace_shim_tile)
```

Data Movement Connections

These are the connections between the tiles. The data movement connections are represented by *object FIFOs*. These define the sender-receiver(s) tiles connection(s), along with the size and type of the data being moved. These also include the connections involving Memory Tiles, which allow the program to utilize their data distribution/joining capabilities.

The data flow mechanism for the object FIFOs follows an acquire and release model. This means that in order to allocate memory to store and read data sent by another tile, a method called *acquire* is called, which requires the object FIFO and the number of objects it will use to allocate the correct memory size. Once the data is used by the consumer, the function *release* is called to deallocate the memory and allow for the next data to arrive. The number of objects the tile is capable of acquiring at once is limited by the total data memory of the tile, as well as the fact that the objects cannot be split between memory banks, so when the application requires to cache data larger than a single memory bank, this must be done in multiple acquire calls.

As shown in Listing 3.3, the obj FIFO takes 4 parameters. The first is the object FIFO name, which has to be unique. Second is the sender, which is fixed to one tile since to combine 2 or more data streams, a Memory Tile needs to be involved as an intermediary. Then there are the receivers, which can be any number of tiles, although these are limited by the total number of locks that are available to the array for a given implementation. After is the object FIFO depth, which can be an integer scalar or an array. These represent the number of objects that will be acquired at any one time by the tiles involved in the connection. When represented as a scalar, it defines the total acquired objects, and if it is an array, it represents the acquired object by each tile in the shape $[sender] + receivers[]$.

Listing 3.3: Object FIFO definition between two tiles

```
depth = 2
dataShape = np.ndarray[(32,), dtype]
obj_fifo = object_fifo("objFifoName", senderTile, receiverTile, depth, dataShape)
```

Core Definitions

These are the definitions of each of the Compute Tiles that will be running in the application. The compiler ties the core definition to its corresponding tile by using its absolute index in the array. From there, it uses the predefined connections to receive, process, and send data to the other tiles. These definitions only apply to the Compute Tiles; the other tiles' behavior is defined implicitly through the object FIFO definitions.

A short example of the core definition can be seen in Listing 3.4. Here, the core is defined by stating the number of iterations the NPU app will run after its definition. This is done through a special loop binding called *range_(N)*. All the other components of the definition need to run inside this loop. After, the core needs to define how many times it will send and receive data to and from its FIFO objects. Finally, has two options for defining computation. If the computation is simple enough and only uses scalar operations, this can be done inside the core definition using Python operators. For most other cases, especially if the vector processor is to be used, the core has to call a kernel containing the desired program. As shown in the example, the kernel uses the data buffers as inputs, which can then be accessed as pointers inside the kernel code.

Listing 3.4: Core definition of Compute Tile

```
kernelFileBin = "kernel.o"
@core(tile, kernelFileBin)
def core_body():
   for _ in range_(ITER_KERNEL): # this needs to be at least the number of
       iterations the app is run
      for _ in range_(ITERS): # The number of times the tile runs per app
          iteration
         input = obj_fifo_in.acquire(ObjectFifoPort.Consume, 1)
         output = obj_fifo_out.acquire(ObjectFifoPort.Produce, 1)
         kernel(input, output, SIZE) # running C++ kernel
         obj_fifo_out.release(ObjectFifoPort.Produce, 1)
         obj_fifo_in.release(ObjectFifoPort.Consume, 1)
```

3.2.3. Kernel Programming

The kernels are programs written in C or C++ that define the behavior of individual CTs of the NPU array. These can be written to use the scalar or vector processors of the AIE-ML as well as any of the supported data types. As seen in Listing 3.4, the objects acquired from the object FIFOs are used as inputs. These can then be accessed inside the kernel as regular C pointers. However, the functions used have to be tied to supported hardware functionality. For instance, it is not possible to print or perform other regular IO operations given the hardware limitation. These incompatibilities are handled by the compiler.

Peano

Peano is an open-source LLVM compiler back-end for the AMD/Xilinx AIE processors, specifically for Phoenix and HawkPoint hardware [53]. It also supports the XDNA2 hardware for the newer models [53]. Peano enables the use of kernels to run in the CTs by providing a compatible compilation target for generating the compiled hardware binary. The generated binaries are then combined with the MLIR-AIE NPU code and are combined into the *xclbin* file, as seen in Figure 3.4. The way the kernels are compiled is by using the *clang*/*clang*++ compiler with the required Peano flags. In cases when the kernel requires multiple files for compilation, these must be compiled separately and then combined into a single static library file (.a). This file can then be used as the kernel binary for the core definition, seen in Listing 3.4.

Vector Library

As previously stated, the kernel called in the CTs core definition can be written in plain C++, and by extension, C. However, to get full use of the AI Engines vector capabilities, it is necessary to use the AIE API header library [18]. This library allows the user to load, process, and store vectorized data inside the user-defined kernels [54]. It provides abstracted types that hide some of the hardware intricacies and facilitate the development process. There are 2 ways vectors can be defined. The first is by loading them from the acquired data objects, as seen in line 6 of Listing 3.5. The second is by broadcasting or populating the vector with data generated inside the kernel, as seen in line 7 of Listing 3.5. After the vectors can be used for computation and finally stored in the output memory space; in the case of Listing 3.5, the input data *in* of size N is loaded in chunks of 64, added 1 to each element, and then stored in *out*.

```
#include "aie_api/aie.hpp" // Vector library
const int VEC_SIZE = 64; // Size of the working vectors
extern "C" {
void add_kernel(bfloat16 *in, bfloat16 *out, uint32_t N) {
    for (int i = 0; i < N; i += VEC_SIZE) {
        auto input0 = aie::load_v<VEC_SIZE>(in + i);
        auto input1 = aie::broadcast<VEC_SIZE>(1.0);
        auto result = aie::add(input0, input1);
        aie::store_v(out + i, result);
    }
}
} // extern "C"
```

Listing 3.5: Compute tile kernel definition

Several types of vector functions make use of the vector hardware, but the most notable are:

- **Arithmetic**: add, sub, mul, abs, etc.
- **Comparison**: eq, gt, lt, max, min, etc.
- **Bits**: bit_and, bit_or, bit_xor, bit_not, etc.
- **Reduction**: reduce_add, reduce_max, reduce_min.
- **Lookup Tables**: lut, lookup, etc.

Compute Tile Stack Sizing

The kernel program in each of the CTs has a default stack size of 1024 bytes [55]. However, if this is not enough, the program may encounter issues with accuracy because the lack of space may cause the stack to be overwritten during the execution of the kernel. This can be avoided by specifying the stack size in the core definition as seen in Listing 3.6, where the size is specified in hexadecimal, so the new stack size is 2560 bytes. The extra memory space is taken from the Program Memory inside the AIE. The stack size is something that must be optimized if program size is a restriction for the application at hand.

Listing 3.6: Core definition of Compute Tile with custom stack size

```
kernelFileBin = "kernel.o"
@core(tile, kernelFileBin, stack_size=0xA00) # this is equal to 2560
def core_body():
   for _ in range_(ITER_KERNEL):
      # the rest of the core definition is the same as the regular definition
```

3.3. Ryzen AI Software

The Ryzen AI Software (Ryzen AI SW) is a software toolchain developed by AMD for optimizing and deploying AI inference on Ryzen AI-powered devices [56]. The development flow requires minimum knowledge of the Ryzen AI hardware, making the switch as seamless as possible for AI developers. However, this toolchain offers less fine-grained control compared to the MLIR-AIE toolchain. The development stack in Figure 3.5 shows the software transformations for running an application using the NPU. Ryzen AI SW allows for the execution of AI models on any of the devices of the Ryzen AI system, whether it be the CPU, the iGPU, or the NPU.

The untrained model is first written with PyTorch or TensorFlow [57]. Then, the model must be converted to the ONNX format, which is compatible with the Ryzen AI SW workflow. Once the model is in the ONNX format, there is the choice of quantizing the model using a Quantizer. Quantization is the process of reducing the precision of digital signals, typically from a higher to a lower precision format [58]. In machine learning models, this is done to convert the weights and activation values to a lower precision data type with the objective of compressing the model size while attempting to maintain the precision of the model. Model deployment is done with ONNX Runtime, which converts the ONNX model description into machine language for running in the target device.

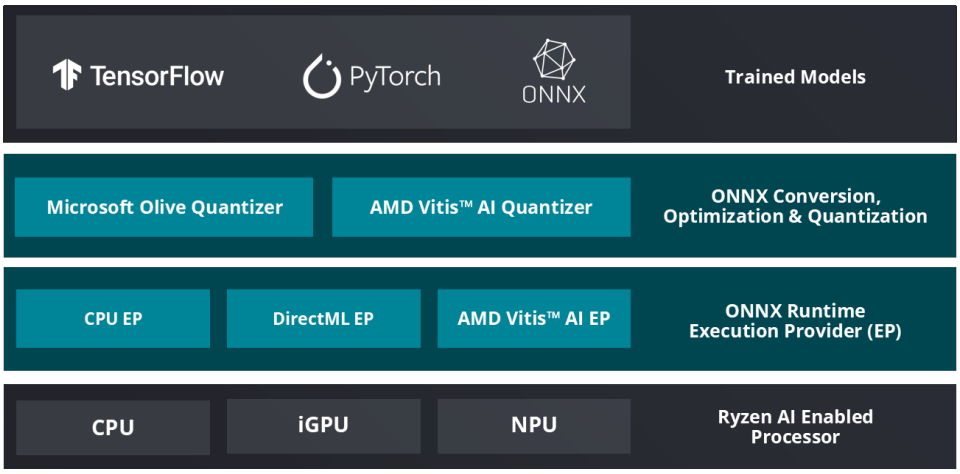


Figure 3.5: Ryzen AI Software development stack, obtained from [56]

3.3.1. PyTorch (Neural Networks)

PyTorch is a deep learning [59] Python library developed by Facebook [39]. One of the most essential components is the *torch.nn* modules, which provide the building blocks for implementing NN applications in a modular and reusable way. The module also provides the ability to define personalized *torch.nn* modules (*nn.Module*), which serve to encapsulate the application and also define behavior not present in the default modules.

The *nn.Module* has two parts in its definition, the initialization function, which is called when the module is instantiated and formats the model using the arguments, and the *forward* function, which defines how the input data flows through the network [39]. For multilayered networks, the *forward* function acts recursively, passing through each of the layers following the graph structure of the module. A simple PyTorch module definition can be seen in Listing 3.7, which defines a simple NN with one input, one hidden, and one output layer.

Listing 3.7: Example implementation of a PyTorch module, obtained from [60]

```
import torch
import torch.nn as nn
class SimpleNet(nn.Module):
   def __init__(self):
      super(SimpleNet, self).__init__()
      self.fc1 = nn.Linear(784, 128)
      self.relu = nn.ReLU()
      self.fc2 = nn.Linear(128, 10)
   def forward(self, x):
      x = self.fc1(x)
      x = self.relu(x)
      x = self.fc2(x)
      return x
```

3.3.2. ONNX

The Open Neural Network Exchange (ONNX) is an open-source format developed to improve the compatibility of AI models with different platforms. It was created by Microsoft and Facebook in 2017 [61], and has become the standard format for representing ML models, allowing them to be written in one framework and then deployed in another, as long as it supports ONNX Runtime. The advantages of using ONNX for representing ML models are reducing development costs for multiple platforms, better implementation integration, and faster model innovation [61]. ONNX Runtime is a high-performance engine for executing ONNX models [61]. It offers support for Windows, Linux, MacOS, and mobile

devices [61]. ONNX Runtime also offers model optimizations depending on the target platform and model type.

3.4. TINA

TINA is a novel framework for implementing non-NN (Neural Network) signal processing algorithms on AI accelerators [13]. The idea is to map mathematical and logical functions to a series of NN layers. TINA makes use of the optimization for NN layers in AI hardware for non-NN applications, as well as ensuring their portability to any AI accelerator platform [62]. Another advantage of TINA is that the non-NN algorithms can then be optimized using methods designed for NNs, such as automatically optimized quantization or training libraries.

TINA is composed of two parts: the basic building blocks and the APIs that use the building blocks to implement the non-NN functions. The NN layers used in the TINA framework as building blocks are:

- Standard Convolution

$$O(h, w, c_{out}) = b(c_{out}) + \sum_{c_{in}}^{C_{in}} \sum_{m}^{M} \sum_{n}^{N} I(h + m, w + n, c_{in}) \cdot K(m, n, c_{in}, c_{out})$$

- Depthwise Convolution

$$O(h, w, c_{out}) = b(c_{out}) + \sum_{m}^{M} \sum_{n}^{N} I(h + m, w + n, c_{out}) \cdot K(m, n, c_{out})$$

- Fully Connected Layer

$$O(c_{out}) = b(c_{out}) + \sum_{c_{in}}^{C_{in}} I(c_{in}) \cdot K(c_{in}, c_{out})$$

- Pointwise Convolution

$$O(h, w, c_{out}) = b(c_{out}) + \sum_{c_{in}}^{C_{in}} I(h, w, c_{in}) \cdot K(c_{in}, c_{out})$$

With TINA, several arithmetic functions have been implemented, including element-wise multiplication, matrix multiplication, element-wise addition, and summation. Also, some common DSP functions have been implemented in TINA, such as DFT (Discrete Fourier Transform), IDFT, and FIR filter. The performance obtained from these implementations shows TINA to be a competitive alternative to other programming frameworks like CuPy [63] or JAX [64]. The commonality between TINA, CuPy, and JAX is their ability to provide a unifying programming framework for running applications in hardware acceleration by establishing an application format that leverages the characteristics of said platform.

4

Solution Designs

This chapter examines various solutions that employ different tools and heuristics to maximize the potential of the Ryzen AI system for the All-Sky Imaging application. For each of the solutions, the overall idea of the solution is described, as well as the details that make it distinct from the other implementations. Details about design choices given hardware features and limitations are explained in Chapter 5.

Some characteristics of the All-Sky Imaging algorithm allow for refactoring to fit the NPU's capabilities better, as well as simplifying implementation:

1. A limitation of the NPU is that the NPU does not have the hardware to execute non-linear functions (exp, cosine, sine, etc.). The way each implementation works around this issue is further explained in 5. But the common factor for the solutions to deal with complex numbers (which complicates the implementation) is to split the real and imaginary values with Euler's formula ($e^{ix} = \cos x + i \sin x$) and calculate the sine and cosine values separately. The proof of this refactor can be seen from Equations 4.1 to 4.5.

$$\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}] = -2\pi \frac{f}{c} \left(u \cdot \mathsf{l}_{l_{\text{ix}}, m_{\text{ix}}} + v \cdot \mathsf{m}_{l_{\text{ix}}, m_{\text{ix}}} + w \cdot \mathsf{n}_{l_{\text{ix}}, m_{\text{ix}}} \right) \tag{4.1}$$

$$Img[l_{\text{ix}}, m_{\text{ix}}] = \Re \left\{ \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{k=0}^{N} V_{i,k} \cdot \exp\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}] \cdot j\right)_{i,k} \right\} \downarrow \tag{4.2}$$

$$Img[l_{\text{ix}}, m_{\text{ix}}] = \Re \left\{ \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{k=0}^{N} V_{i,k} \cdot \left(\cos\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}]\right)_{i,k} + j \cdot \sin\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}]\right)_{i,k} \right) \right\} \downarrow \tag{4.3}$$

$$Img[l_{\text{ix}}, m_{\text{ix}}] = \Re \left\{ \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{k=0}^{N} \left(\Re\{V_{i,k}\} + j \cdot \Im\{V_{i,k}\} \right) \left(\cos\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}]\right)_{i,k} + j \cdot \sin\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}]\right)_{i,k} \right) \right\} \downarrow \tag{4.4}$$

$$Img[l_{\text{ix}}, m_{\text{ix}}] = \frac{1}{M \cdot N} \sum_{i=0}^{M} \sum_{k=0}^{N} \Re\{V_{i,k}\} \cdot \cos\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}]\right)_{i,k} - \Im\{V_{i,k}\} \cdot \sin\left(\mathcal{A}[l_{\text{ix}}, m_{\text{ix}}]\right)_{i,k} \tag{4.5}$$

2. As an optimization with the architecture of the NPU, matrices (2d vectors) can be flattened to work with 1D vectors, given that all the operations are element-wise. This gives more flexibility for the implementation while maintaining the correctness of the algorithm. In theory, this also works when increasing the dimensions of the inputs, as long as the change is congruent among them for the type

of operations. With this idea, the algorithm is refactored to the Formula 4.6. This is the version used for all the MLIR-AIE implementations.

$$\begin{aligned} \mathcal{A}[p_{\text{ix}}] &= -2\pi\frac{f}{c}\cdot\left(u\cdot l_{p_{\text{ix}}} + v\cdot m_{p_{\text{ix}}} + w\cdot n_{p_{\text{ix}}}\right) \\ Img[p_{\text{ix}}] &= \frac{1}{N}\sum_{i=0}^{N} \Re\{V_i\}\cdot\cos\left(\mathcal{A}[p_{\text{ix}}]\right)_i - \Im\{V_i\}\cdot\sin\left(\mathcal{A}[p_{\text{ix}}]\right)_i \end{aligned} \tag{4.6}$$

3. Another assumption that simplifies the implementation of the algorithm is fixing the number of antennas to 96. This, in turn, sets the size of the baselines and the visibilities to be $96 * 96 = 9216$ elements each. This number was chosen because this is the maximum number of antennas in the core and remote stations when separating LBAs and HBAs. These two antenna types cannot be sampled using the same frequency, so there is no case where these would need to be used together. By fixing the number of antennas, the input size for the implementations is given by the number of output image pixels. The calculation for input data size can be obtained with Equation 4.7.

$$InputBytes = (1 + 5 * 96^2 + 3 * npix\_l * npix\_m) * DataTypeBytes \tag{4.7}$$

We explore four different design approaches for the application:
0) MLIR-AIE Pipelined
1) MLIR-AIE Parallel
2) MLIR-AIE Bi-Pipelined
3) TINA

One important detail about the MLIR-AIE implementations is the switch from using float typing to using bfloat16. This is primarily due to the incomplete support for the float type in the architecture, and switching to this type (theoretically) doubles the potential performance in terms of computation, data caching, and data movement.

On the other hand, TINA was used to implement a single solution. In turn, this involves further refactoring the modified algorithm in Equation 4.5, which is explained in Section 4.4. The solution description includes the new algorithm and the mathematical proof of its correctness. A high-level PyTorch implementation of the algorithm can be found in Appendix A.

4.1. Solution 0: MLIR-AIE Pipelined

Idea: This may also be called the ”naive” solution since it attempts to map the All-Sky Imaging algorithm to the NPU as directly as possible. The idea is to separate the operations done in the inner loop of the algorithm, as seen in Listing 2.1 (with the modifications seen in Equation 4.6), and execute them in a pipeline structure using the Compute Tile to spread out the computation.

Solution Rational: The reason for choosing this solution strategy has to do with the NPU hardware platform. A pipelined solution takes advantage of the dataflow execution model by maximizing the data movement capabilities while distributing the computation among the CTs of the NPU. The goal of Solution 0 is to leverage this hardware architecture as much as possible in an attempt to maximize throughput. Furthermore, the VLIW and vector processors in the CTs provide an opportunity to reduce the latency of the individual pipeline stages. This solution serves as a baseline for the other MLIR-AIE implementations since it is structured as the expected utilization of the NPU device as described in [43].

Figure 4.1 shows the mapping of the algorithm to the NPU tiles, where the tiles using the same colors run in parallel and the names of the kernels are written under the tiles’ axes. This diagram shows the data movement between tiles during the entire run of the All-Sky Imaging algorithm, including the initial data distribution for caching in the CTs. The arrows show the data transfers between tiles, including the data transfer sizes. For instance, the connection between tiles CT01 and CT12 is done in 2 movements of 4608 elements from CT01 to CT12.

Equation 4.6 shows the order of the operations inside the loop. These can be executed in forward-feeding Compute Tiles, thus maximizing the throughput. The exceptions are the sine and cosine functions and the subsequent multiplication with the visibilities vectors. These can be executed in parallel up to the point of the subtraction of the real and imaginary components, where they merge into a single pipeline channel once again.

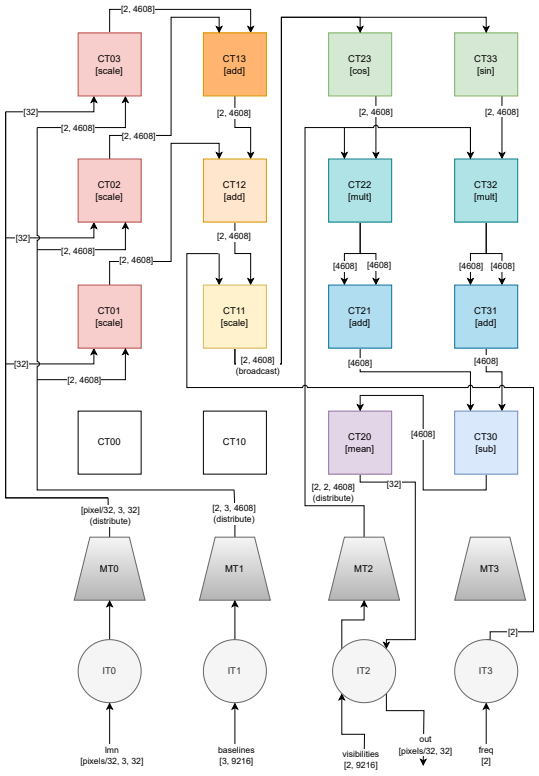


Figure 4.1: Array implementation of the MLIR-AIE Pipelined solution.

The order of operations is kept almost the same in comparison to the base algorithm. The exceptions are the parallel execution of the trigonometric functions and the multiplication of the baselines with their respective cosine directions. The reason for doing these operations in parallel instead of strictly staying with the pipeline model will be further discussed in Chapter 5. At the beginning of execution, the input data for the frequency, the baselines, and the visibilities data are distributed using Memory Tiles to their respective Compute Tiles.

After the initial data distribution, the only data movements taking place from the Shim Tiles are the values of l, m, and n. These are sent in chunks of 32 elements to comply with the 32-bit multiple requirement for data movement. From there, the data flows throughout the array until arriving back into the output Shim Tile. The input and output chunk sizes are shown to be 32 elements wide in Figure 4.1; however, the calculations utilizing each of these elements do not happen in parallel, they are accumulated at the end to meet the chunk size requirement, and ultimately sent out. The reasons for choosing this chunk size are further explained in 5.

The most obvious downside of this strategy is the fact that data is moved between tiles throughout the execution. Because of the conversion to bfloat16 between tiles, instead of relying more heavily on accumulators for intermediate results, this has the potential to reduce the accuracy of the output.

4.1.1. Performance Prediction

Figure 4.2 shows a simplified version of the solution where the tiles working in parallel are seen as a single pipeline stage with the name of the kernel and the pipeline stage index. The number under the name of the kernel is the number of tiles running for this stage. With the current pipeline design, 9 stages are working in parallel once the pipeline is filled. This gives a theoretical speedup of 9. This is on top of the vector and VLIW capabilities of the CTs, and the parallelization of some of the stages.

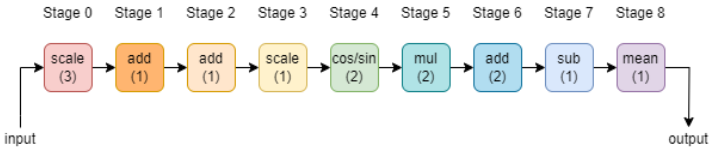


Figure 4.2: Pipeline design for Solution 0

4.2. Solution 1: MLIR-AIE Parallel

Idea: Given that most of the calculations are independent, the operations in the inner loop are fully parallelizable. Thus, these operations can be distributed across N CTs, and then the results can be combined to calculate the mean for each pixel. Figure 4.3 shows the tile distribution for this design. The brackets in the diagram signify connections to multiple tiles to avoid overcrowding in the figure, and the arrow at the end shows the direction of said connection.

Solution Rational: The justification for this solution strategy is that the systolic array structure of the Ryzen NPU allows for the even distribution of computation of the algorithm. This, paired with the vector processor in the CTs and the parallelization opportunities of the All-Sky Imaging Algorithm, suggests that an implementation using Solution 1 has the potential to achieve a high degree of acceleration compared to the baseline.

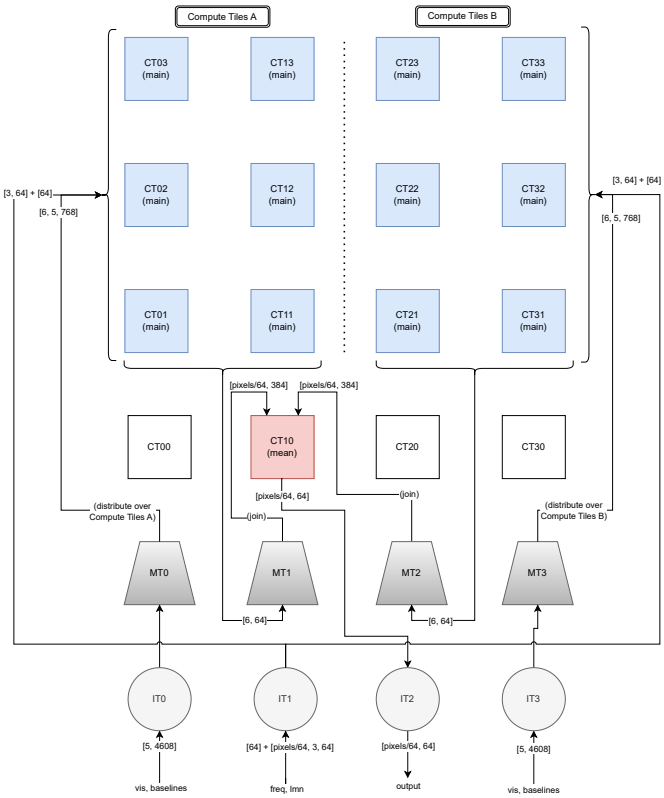


Figure 4.3: Array implementation of the MLIR-AIE Parallel solution.

A clear advantage of this design is that it is highly scalable when porting to an NPU with more Compute Tiles. As long as the input data can be evenly split over the number of tiles, the implementation changes are few. So this parallel design has a theoretical speedup of N, for N tiles the hardware allows to distribute the data over. If the number of tiles cannot evenly split the input data, this can be padded

with zeros to achieve the same result. This assertion is theoretical, and testing is required to confirm its effectiveness.

Given the limit of 6 input/output streams of the MTs, the parallel computing is done over 12 CTs. The tiles are split into 2 groups, CTs A and CTs B. For each group, 2 Memory Tiles are needed. One MT to distribute the input data, and one to then join the output streams and feed into the *(mean)* tile at the end. The *(mean)* tile takes the partial results of all 12 tiles and calculates the final average of the inner loop.

Each of the CTs labeled as *(main)* runs the same kernel with the data of the baselines, visibilities, and frequency ”cached” in their respective memory banks. This is possible because, as mentioned in Section 2.1.4, the computations for each of the elements are independent of each other until the mean aggregation. With this, the calculation for a single pixel can be split evenly over the 12 *(main)* tiles.

Unlike the previous design, the chunk size for the cosine directions (l, m, n) is 64. The kernel inside the *(main)* tiles iterates over data chunks of the cosine directions and does the computation for 64 pixels, which are then sent to the respective MT for joining. This is because of the way the *(mean)* tile is set up, where the working vector size and the chunk size must match.

One particular advantage of this design compared to the others is that the LUTs are distributed evenly over the working CTs. Given that the rate of execution for this type of operation is lower compared to the other arithmetic vector operations, it can be expected that the execution time will be reduced significantly. One limitation of this design, however, is the lower number of compute tiles utilized because of the limit of distribution/join channels in the MTs.

4.2.1. Performance Prediction

With this implementation, the theoretical speed-up is expected to be given by the number of CTs the data can be distributed. In this case, the theoretical speed-up is the number of *(main)* tiles, which is 12. Additionally, since most of the calculation is done with a single kernel, and since the output only requires the real component of the result, when the value of n is imaginary, the entire calculation can be skipped, thus reducing the latency of the overall calculation. Since this happens for 21.46% (see 2.1.4), the implementation experiences approximately 1.27 additional speedup. Furthermore, the implementation’s speedup would also be affected by the vector processing and the low number of data movements.

4.3. Solution 2: MLIR-AIE Bi-Pipelined

Idea: This solution attempts to combine both ideas of the previous solutions to pipeline the operations, while at the same time parallelizing the LUT operations. The number of tiles restricts the idea to be a two-channel pipeline version of Solution 0. Figure 4.4 shows the tile layout of the design. In this diagram, some connection lines are also swapped for lines using small figures to signal a connection between tiles. For instance, IT1 and CT01 have a connection signaled by both connection lines starting/ending with a blank square. In this case, the connection flows from IT1 and CT01. The other connections use other figures to avoid confusion.

Solution Rational: The rationale for this solution strategy is to leverage both the dataflow capabilities of the NPU, while at the same time achieving a degree of parallelism from the vector processors and the distribution of computation over the CTs. This serves as an attempt to take advantage of all the capabilities of the Ryzen NPU hardware to maximize throughput and minimize the latency of a potential bottleneck in the implementation. Being a middle point between Solutions 0 and 1, this solution also gives an idea of the benefits of both strategies and which one results in better performance.

This design fully uses the available CTs by having most of the operations be split into the two channels up to the point of the subtraction of the real and imaginary components of the calculation. Similar to Solution 0, the completion of a single pipeline iteration outputs pixels in chunks of 64, but the calculation of each of them is done sequentially with respect to one another. By maximizing array utilization and better balancing the pipeline stages, it is expected to increase the performance in comparison to the more ”naive” version.

The functionality of the CTs for this solution can be split into three groups. The first two are Compute Tiles A and Compute Tiles B, which are analogous to the pipeline tiles of solution 0. These work with the same program kernels, but the data is distributed over the two channels; therefore, the compute load is halved. The other tile group is composed of the tiles labeled (*sub*) and (*mean*), which take the

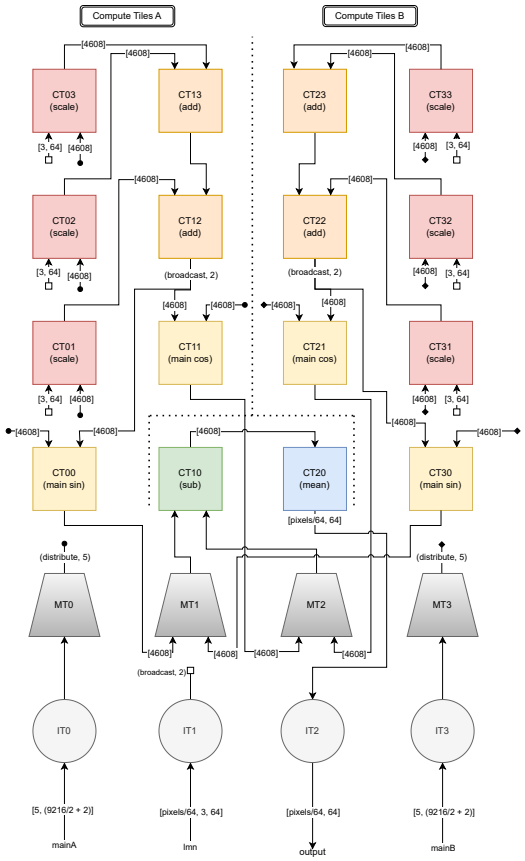


Figure 4.4: Array implementation of the MLIR-AIE Bi-Pipelined solution.

joined data streams of both channels and finish the execution in a single pipeline. The chunk size for sending the cosine direction values is also 64 for this solution.

4.3.1. Performance Prediction

With this implementation, the theoretical speed-up is given by two factors: the two channels and the number of pipeline stages. Given that both channels are computed in parallel, in terms of performance, the design can be seen as a 6-stage pipeline. Figure 4.5 shows this pipeline design with the numbers under the kernel name representing the number of parallel tiles in this stage. In spite of having fewer pipeline stages when compared to Solution 0, the performance can be expected to be better because of having a better balance between them. By splitting the LUT operations into two channels, which are expected to cause the bottleneck, the expected improvement is approximately 2x when compared to Solution 0.

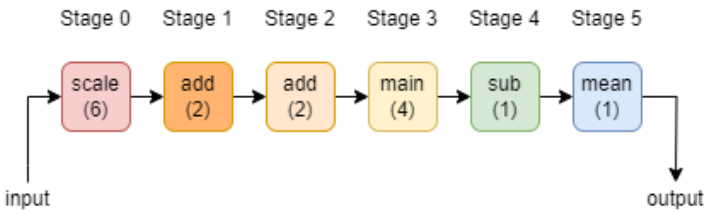


Figure 4.5: Pipeline design for Solution 2

4.4. Solution 3: TINA

Idea: Using TINA and Ryzen AI SW leads to a more restrictive implementation due to the types of operations that are allowed to be used. Because of this, the algorithm implemented for this solution has been reduced to perform the minimum functionality of the imaging application. In this version, the multiplication of the baselines with the cosine directions and the exponential function is precomputed. This is because, as previously mentioned, the hardware cannot run non-linear functions, but unlike IRON, there is no direct alternative. These operations are then refactored to use NN layers instead of matrix/vector operations.

Solution Rational: The justification for using this solution strategy is that the mapping of the algorithm to use NN layers rather than the matrix operation leverages the features in the NPU used to maximize the performance of ML workloads.

This solution also does not do the normalization of the pixel's output intensity. This is not a significant drawback for an imaging application since the value range and magnitude are not relevant for common plotting applications/libraries like Matplotlib [65]. As long as this fact is taken into account when computationally comparing implementation outputs, this should not affect the diagnostic purpose of this solution.

Although this version of the application gives less flexibility than the IRON-based solution, in a day-to-day scenario, the application would still be able to perform its function. As long as the image size, antenna set, and sampling frequency stay constant, this version can be used without the need for re-compiling it. This is in exchange for better potential speedup given its reduced amount of computations.

The pre-computed matrix will be referred to as the ”exponential matrix”. This creates a tensor that serves as input for the weights of the PyTorch model. During runtime, the visibilities are convolved with the matrix to generate each of the pixels of the image.

4.4.1. Proof

In order to implement this formula within TINA, we use one of the basic building blocks of TINA, the convolution.

$$O(h,w,c_{out}) = b(c_{out}) + \sum_{c_{in}}^{C_{in}} \sum_{m}^{M} \sum_{n}^{N} I(h+m, w+n, c_{in}) \cdot K(m,n,c_{in},c_{out}) \tag{4.8}$$

We start by setting the weight matrix K equal to the complex exponential matrix generated by the sky image equation.

$$K\left(l,m,n,\ c_{\text{in}}=1,\ c_{\text{out}}\right) = \exp\left[-2\pi i\left(u_{m+1,n+1}\,\mathsf{l}(c_{\text{out}}) + v_{m+1,n+1}\,\mathsf{m}(c_{\text{out}}) + w_{m+1,n+1}\,\mathsf{n}(c_{\text{out}})\right)\right] \tag{4.9}$$

Then, by creating two input channels, one for the real part of the weights and one for the imaginary part, and by setting the bias to be a zero vector, we get the following equation:

$$O(h,w,c_{out}) = \sum_{m}^{M}\sum_{n}^{N} \Re(I(h+m,w+n)) \cdot \Re(K(m,n,c_{out})) + \sum_{m}^{M}\sum_{n}^{N} \Im(I(h+m,w+n)) \cdot \Im(K(m,n,c_{out})) \tag{4.10}$$

Now, by multiplying the imaginary weight by -1, we get the following formula

$$O(h,w,c_{out}) = \Re\Big(\sum_{m}^{M}\sum_{n}^{N} I(h+m,w+n) \cdot \exp\left[-2\pi i\left(u_{m+1,n+1}\,\mathsf{l}(c_{\text{out}}) + v_{m+1,n+1}\,\mathsf{m}(c_{\text{out}}) + w_{m+1,n+1}\,\mathsf{n}(c_{\text{out}})\right)\right]$$
$$- \Im\Big(\sum_{m}^{M}\sum_{n}^{N} I(h+m,w+n) \cdot \exp\left[-2\pi i\left(u_{m+1,n+1}\,\mathsf{l}(c_{\text{out}}) + v_{m+1,n+1}\,\mathsf{m}(c_{\text{out}}) + w_{m+1,n+1}\,\mathsf{n}(c_{\text{out}})\right)\right]\Big) \tag{4.11}$$

making it equivalent to equation (4.5), except for the normalization.

4.4.2. Performance Prediction

It is difficult to predict the performance increase for this solution, given the black-box nature of the Ryzen AI SW toolchain. However, it is expected to have a noticeably short execution time compared to the baseline, given the algorithm's reduction. The number of operations is reduced by less than half as a result of calculating the exponential matrix at compile time. With this model optimization and the use of the NPU hardware, the expectation is that this solution results in the lowest execution time out of the NPU implementations.

5

Implementation

This chapter delves into the practical details of the implementation process, focusing on the decisions made based on the technical limitations and opportunities of the hardware and tooling at hand. In particular, the features of the implementations are described in more detail, and the rationale for each is explained. Key areas include memory constraints, performance optimizations, and integration with the correct toolkit versions. The chapter shows how specific hardware constraints and available tools directly shaped these implementations.

5.1. MLIR-AIE Implementations

5.1.1. Trigonometric Functions

As mentioned in Chapter 3, the AI Engines do not have the hardware to calculate non-linear functions. There are two options for how this limitation is usually solved. The first is a mathematical approximation of the functions, such as in [66], where the implementation accelerates the approximation of the sine function for an embedded application. The second option is to use LUTs for each of the required functions and access them during run time, requiring less or no computation. The option for mathematical approximation is usually for cases where the platform has dedicated hardware for its calculation or does not have sufficient memory for LUT data storage. This is not a good fit for this use case because the objective is to minimize execution time and adding computation would go against this objective. Also, this hardware is not limited in memory to the point of discarding the use of LUT. Therefore, given that LUTs are supported by the toolkit, this method was the chosen strategy.

Given the refactoring to switch from using the exponential function to using sine and cosine, two LUTs are needed. The sine and cosine functions are periodic functions commonly used in DFT applications. The periodicity of the functions makes it possible to only implement the LUT for the range of the period, which in this case is $[0, 2\pi]$. The others can be obtained through a truncation functionality for the inputs of the LUT for higher positive values. For negative values, the output can be obtained through two trigonometric identities:

- Reflective Cosine Identity: $cos(-\theta) = cos(\theta)$
- Reflective Sine Identity: $sin(-\theta) = -sin(\theta)$

To reduce inaccuracies caused by low LUT granularity, while minimizing program size for the kernels using them, the LUT size was set to 512 elements. For instance, this means the value for $cos(0)$ would be located at index 0 of the table, and the value for $cos(2\pi)$ at index 511. This means the granularity for the angle input is of $2\pi/512$. During implementation, any increase in the LUT size beyond 512 did not affect the accuracy of the implementation, although more testing is needed to confirm the effects of the LUT size on the accuracy of the output.

To call the LUT lookup function, the input values have to be mapped to the correct LUT index. For cosine, this can be done directly since the output is the same for the positive or negative version of a given angle θ. For sine, the sign can be extracted using bit-wise vector operations and reattached to the output. When the sign of the angle is extracted, the value must be multiplied by the inverse of

the granularity factor ($512/2\pi$) to map it to the correct index. For this implementation, given that the working datatype is bfloat16, it needs to be converted to an integer datatype for the LUT table to use as an index. The Vector Unit hardware can convert 16 lanes of bfloat16 to int32 and vice versa.

The LUT functionality of the toolkit is limited to a 32-vector lane access size. At the same time, the access is done in chunks of 4 parallel lookups. Both of these factors place a bottleneck in the implementation performance, which is further discussed in Section 6. The limit in vector size for LUT access causes conflict with kernels that use higher vector sizes. However, this can be overcome by using the $extract$ and $concat$ functions of the vector library. As seen in Listing 5.1, a vector with a size of 64 elements can be split into 2 vectors of 32 elements. From there, the LUT table operations can be called, and then the outputs can be concatenated to have the correct output size.

```
// Getting the angle vector
auto theta = aie::load_v<64>(in);
// Extracting sub-vectors
auto theta0 = A.extract<32>(0); // first 32 elements of the theta vector
auto theta1 = A.extract<32>(1); // last 32 elements of the theta vector
// Calling LUT sine function
auto sin0 = sin_bfloat16(theta0);
auto sin1 = sin_bfloat16(theta1);
// Concatenating results
auto sin = aie::concat(sin0, sin1);
```

Listing 5.1: Extraction and concatenation of sub-vectors for LUT operations

In order to achieve 4 parallel lookups of the LUT, the hardware requires the data to be duplicated said number of times. For LUTs of 512 elements, this is done with 2 arrays of 1024, for which the elements are repeated once every 128 elements (this is also a requirement according to the documentation [47]). This results in something similar to Listing 5.2.

```
alignas(aie::vector_decl_align) bfloat16 sin_ilut_ab[1024] = {
    // Array content
}
alignas(aie::vector_decl_align) bfloat16 sin_ilut_cd[1024] = {
    // Array content, equal to sin_ilut_ab
}
```

Listing 5.2: Structure for 4 parallel lookup of 512 element LUT for the sine function using bfloat16 datatype

5.1.2. Input Formatting and Data Caching

In all of the MLIR-AIE designs, most of the input data is cached inside the memory banks of the CTs. This is so that the data can be reused for every pixel of the calculation. Otherwise, the input data would need to be duplicated for every pixel, given that the NPU cannot iterate multiple times over the same buffer object. In this case, the input data cached for all designs are the visibilities, the frequency, and the baselines. The cosine directions are streamed throughout the execution of the application with every iteration. This need for data caching influenced the way the input data had to be formatted to fit into the CTs' memory banks.

Implementation 0: MLIR-AIE Pipelined

For the implementation of Solution 0, the data caching is the most complicated out of the three since it requires splitting the input data streams into multiple memory banks for many of the CTs. In particular, for the CTs getting the initial input of the baselines and visibilities, these must cache a total of 9216 bfloat16 (18432 bytes), which exceeds the size of 16 KB of the memory bank. As mentioned in Chapter 3, data that exceeds the size of a single memory bank cannot be acquired in one call, since the object cannot be split between memory banks.

The solution for the size limit of the memory bank is to acquire the data in two calls and thus create 2 memory objects of 4608 instead of 1 of 9216. A side effect of this is that the kernels used in these tiles are called more than once to process each of these objects. Following the diagram in Figure 4.1, the tiles that follow this data acquisition pattern are the tiles CT01, CT02, CT03, CT22, and CT32. This also maps to the number of data objects that are moved during the execution of each iteration, where these tiles have to transfer the two objects separately to the next pipeline stage.

Another cached input for this implementation is the frequency, which is a single bfloat16 scalar. However, the streaming architecture of the NPU has a minimum data transfer size of 32 bits [47], so the frequency input had to be padded with another bfloat16 to meet this requirement. This, in turn, explains the data length in 4.1 for IT3, where the 2 matches this padded input.

This data distribution scheme affects the way the data must be sent to the NPU device. The data inputs are initially split into 4 elements: the visibilities, the baselines, the cosine directions, and the sampling frequency. Since the number of inputs of the base algorithm matches the number of NPU inputs, these just need to be rearranged to match the input format. As seen in Figure 5.1, the frequency needs to be padded to be usable as an input. The visibilities are split into two chunks of 4608 elements, both for the real and imaginary components, and these are alternated as [4608(Real), 4608(Imag), 4608(Real), 4608(Imag)]. Similarly, the baselines are split into chunks of 4608, and these are alternated as [4608(u), 4608(v), 4608(w), 4608(u), 4608(v), 4608(w)].

The cosine directions are formatted in a way that the values for l, m, and n are also arranged interleaved in chunks of 32, in other words, 32 values of l, 32 values of m, 32 values of n, and so on. This is so these can be distributed to the tiles that multiply them with the baselines, where every iteration distributes 96 values to 3 tiles. The reason for using a chunk size of 32 and not 64 like the other two MLIR-AIE implementations is because of the way the core definitions were written.

For the tiles that use the cosine directions, the core definition iterates over them and calls the kernel using the index of the value as an input. The core definitions do not translate regular Python *for in range(n)* calls as loops. This is done via a special Python binding called *range_* as seen in 3.4. However, this special function does not provide the index of the iteration; thus, *range* was used. The issue is that the *range* function gets compiled to MLIR-AIE as a direct repetition of the content of the loop. Using a chunk size larger than 32 caused errors in the implementation caused issues at the moment of compilation. Therefore, this was kept at 32. Despite this, there was no observed benefit to optimizing the data transfer chunk size for the cosine directions for the MLIR-AIE applications, so the current numbers are somewhat arbitrary, but also to avoid using more memory at once than necessary.

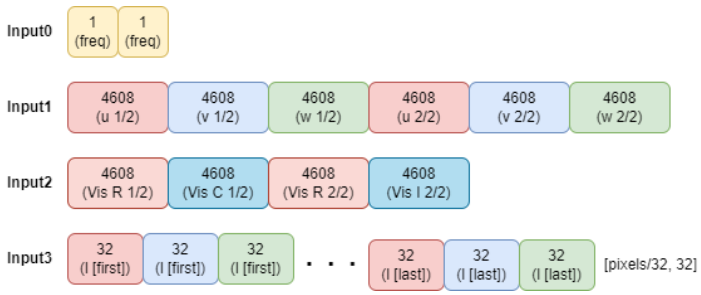


Figure 5.1: Input format for Implementation 0.

By formatting the input to correctly match the NPU implementation, the input data size is increased slightly compared to the Python baseline (see Listing 2.1.4). The changes only include the frequency input padding, which results in a new input size of 124932 bytes + pixels*3*2.

Implementation 1: MLIR-AIE Parallel

For Implementation 1, the data caching is much more flexible, given the relatively small amount of data needed to stay in the CTs during the program's execution. In this case, the data for the baselines and the visibilities are evenly split over the main CTs (see Figure 4.3). Also, the frequency data is broadcast to the main tiles and cached during the run of the application.

The data is streamed through 3 input streams. The first is a combination of the cosine directions and the frequency. Given that the data streams must be sent in equally sized chunks, the frequency data must be padded to meet this requirement. Since the chunk size for the cosine direction is 64 for this implementation, the frequency is padded to become this size, as seen in Input 0 of Figure 5.2. The rest of the input is the data from the cosine directions formatted in the same fashion as in Implementation 0. The chunk size of 64 was chosen arbitrarily since the size of the transfer data for the cosine directions did not appear to affect the performance of the application.

The other two inputs are made from the data of the visibilities and the baselines. As seen in Inputs 1 and 2 of Figure 5.2, the visibilities and baselines data are split into two. Each half is assigned to an

input stream and concatenated with the other inputs, such that the input data flows in the order [VisR, VisI, u, v, w], each input containing 4608 elements.

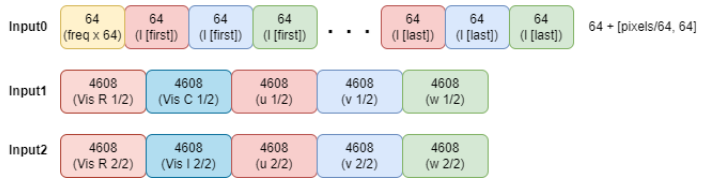


Figure 5.2: Input format for Implementation 1.

The total data cached in each of the main tiles is then calculated as $visSize/N + baselinesSize/N + freqSize$ where $N = 12$, the size of the visibilities is $2 * 9216$ for the real and imaginary components, the size of the baselines is $9216 * 3$ and the size of the frequency is 64. In total, this comes out as 3904 bfloat16 or 7808 bytes for each main CT, which fits nicely inside a single memory bank. With this formatting, the new size of the input data for Implementation 1 is 92288 bytes + pixels*3*2 bytes.

Implementation 2: MLIR-AIE Bi-Pipelined

The data caching scheme of Implementation 2 is similar to Implementation 0. However, given its parallel design, there is no issue with caching the full input since these are distributed over more tiles. In this implementation, the visibilities and baselines are distributed evenly over the two pipeline channels and cached in the respective tiles. This is done through two input streams, each feeding into each tile group.

The frequency data is also cached in the tiles during the run of the application. However, this works differently compared to the other implementations. In this case, the frequency data is used as padding to the 2 main input streams, as seen in Figure 5.3, and to maintain the data transfer size, this is done for each of the data chunks of the input streams of Input 1 and Input 2. Given that these are only 10 bfloat16 (20 bytes) per input, it does not affect the data transfer latency in a significant way.

Figure 5.3 shows the format in which the inputs must be formatted to be fed into the application. To form Inputs 1 and 2, the visibilities and the baselines are arranged in the same way as for Inputs 1 and 2 of Implementation 1, but 2 instances of the frequency are placed before each 4608 data chunk. Input 0, on the other hand, is formatted the same as Input 3 of Implementation 0, with the alternating values of l, m, and n, but in this case, in chunks of 64. The chunk size of 64 for the cosine directions was chosen arbitrarily since it does not appear to influence the performance of the application.

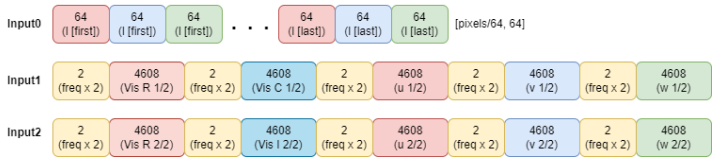


Figure 5.3: Input format for Implementation 2.

The formatting of the input data increases the total size of the input for the application. This is caused by the extra padding to Inputs 1 and 2 for the frequency data. The new size of the input data for Implementation 2 is 92180 bytes + pixels*3*2.

5.1.3. Kernels

The kernels used for the MLIR-AIE implementations were written using C++ and the AIE API for accessing the vector processor and maximizing performance. Optimizing the kernels is the most important aspect of obtaining the best possible performance, so a thorough analysis is necessary to be aware of all the possible avenues for improvement. In particular, the vector size used for vector operations is the main driving factor for this optimization.

Implementation 0: MLIR-AIE Pipelined

Implementation 0 is the version with the most number of distinct kernels in its design. This can be explained by the fact that this implementation has the most pipeline stages, thus distributing the computational tasks among the tiles more evenly than the others. The reliance on the pipeline structure

makes it crucial to optimize the kernels to minimize the latency of the stages and avoid unnecessary bottlenecks. This implementation uses 7 kernel programs, which are used a various number of times along the pipeline. These are summarized in Table 5.1. ”Args” refers to the number of arguments of the kernel function, ”Tiles Used” refers to the number of tiles that call this kernel, and ”Calls/Pixel” refers to the number of times that the kernel is called in the array per pixel.

Kernel	Description	Args	Tiles Used	Calls/Pixel
add	Adds 2 input streams element-wise over N elements and outputs the result.	4	4	8
mean	Computes the mean of an input stream over N elements and saves the result.	3	1	1
mul	Multiplies 2 input streams element-wise over N elements and saves the result.	4	2	4
scale	Multiplies N elements of an input stream by a scalar and saves the result.	4	4	8
sub	Subtracts one input stream from another element-wise over N elements and outputs the result.	4	1	1
cos	Outputs the cosine for N elements of an input stream and saves the result.	3	1	2
sin	Outputs the sine for N elements of an input stream and saves the result.	3	1	2

Table 5.1: Summary of Kernel Functions of Implementation 0.

Each of these kernels can use a wide range of vector sizes, which greatly influences their performance. Thus, choosing the best vector size for each is crucial to achieving the best overall performance for the implementation. The way the vector size was optimized was by running the kernels individually in benchmark applications using all the available vector sizes (within reason). From these results, the version with the lowest execution time is the one chosen. A graph showing this process for Implementation 0 can be seen in Figure 5.4. Here, the kernels have been found to have a clear minimum, shown by the marked dot on the plot line, although some achieve better results from this process than others. In particular, the cos and sin kernels show a low level of performance, which their use of LUT operations can explain. Ultimately, the kernels containing the LUT operations are bound as the bottlenecks of the implementation, given their high execution time in comparison to the rest, in particular the sine kernel.

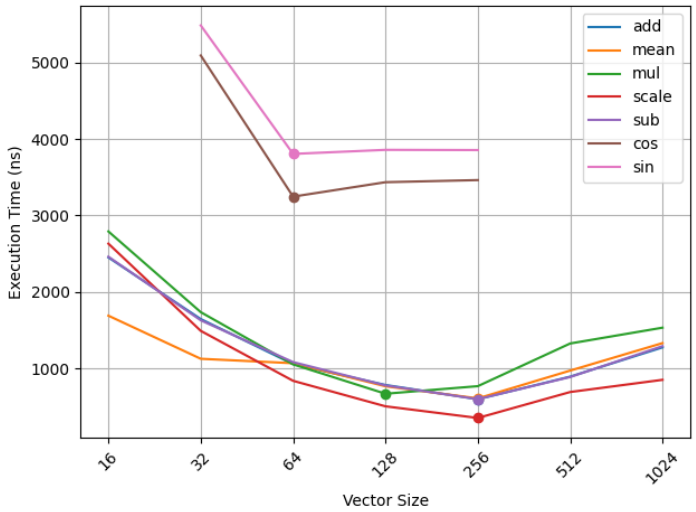


Figure 5.4: Execution time of the kernels from Implementation 0 given by the vector size used in the program.

Implementation 1: MLIR-AIE Parallel

In contrast to Implementation 0, Implementation 1 has the least number of kernels used in its design. There are 2 kernels used in this implementation, which greatly simplifies the development process because it means reusing this for all the tiles used for parallelizing the operations. The kernels used in Implementation 1 are summarized in Table 5.2.

Kernel	Description	Args	Tiles Used	Calls/Pixel
main	Performs most computations of the modified All-Sky Algorithm loop, excluding the final mean. Arguments are frequency, baselines, visibilities, cosine directions, and output pointer.	9	12	$\frac{12}{64}$
mean	Aggregates results from main tiles by computing the mean over 2 input streams (not element-wise) to produce final pixel values.	4	1	$\frac{1}{64}$

Table 5.2: Summary of Kernel Functions of Implementation 1.

Same as with the kernels of Implementation 0, the kernels for this implementation can use many vector sizes in their programs. The optimization for these kernels was also driven by the minimum execution time among different versions, so the kernel used different vector sizes. As seen in Figure 5.5, the main kernel shows a clear minimum for execution time given the vector size, while the mean kernel appears not to be as affected by this factor.

There are some caveats regarding the optimization of the 'mean' kernel. For this kernel, the vector size must be the same as the input data chunk size of the cosine directions; therefore, a smaller vector size would automatically mean a lower execution time. So, to circumvent this feature of the kernel for the optimization process, the execution time was weighted to process the maximum vector size of 1024; in other words, the version with the vector size of 16 was multiplied by 64 to match the same number of computations as the version using 1024-element vectors.

As expected, the main kernel is the bottleneck of the application, even though optimizing the vector size, some performance can be obtained. This is mainly because of the LUT operations, which are proven by the performance of the same kernel but without the LUT operations in Figure 5.5 labeled as "main (w/o LUT)". For this version of the kernel, the execution time is reduced drastically to less than half of the original. Nevertheless, the LUT operations are a necessary part of the implementation, so they cannot be skipped.

Another detail regarding the implementation is that when optimizing the vector size of the main kernel, the best execution time was obtained with a vector size of 128 elements, but when running the entire application, the vector size for the main kernel that rendered the best overall performance was 64, so this is the one ultimately used. This can be explained by the fact that both vector sizes result in a similar performance of the kernel, so when running the kernel along with other operations, the compiler may have caused the full application to favor 64 instead of 128-sized vectors.

Implementation 2: MLIR-AIE Bi-Pipelined

The kernels for Implementation 2 have a similar distribution to the kernels of Implementation 0, but the stages 4, 5, and 6 are merged into one in the kernels named "main". This implementation uses 6 different kernels. This implementation also relies on the optimization of all the kernels to minimize the latency caused by a potential bottleneck. The kernels used in Implementation 2 are summarized in Table 5.3.

As with the previous implementations, the vector size of the kernels was optimized to minimize their execution time. As seen in Figure 5.6, the kernels found a clear optimum in regards to the vector size used for their implementation. The kernels using LUT operations appear to have found the minimum execution time using 64-element vectors, while the rest work best with 256 vectors. This is consistent with the kernels of the other 2 implementations.

Since this is a pipelined implementation, the kernel with the most latency bottlenecks affects the application's performance, which in this case is caused by the "main_sin" kernel. This is because of its use of LUT operations for the sine function needed for its implementation. This is proven by the

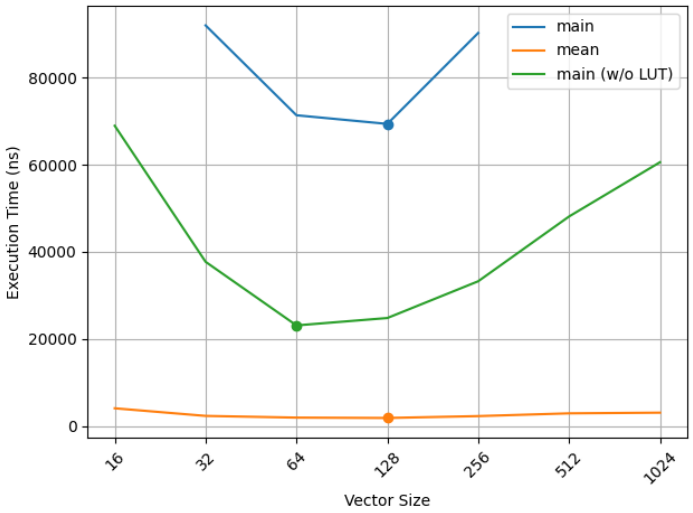


Figure 5.5: Execution time of the kernels from Implementation 1 given by the vector size used in the program.

Kernel	Description	Args	Tiles Used	Calls/Pixel
add	Adds 2 input streams element-wise over N elements and outputs the result.	4	4	4
mean	Computes the mean of an input stream over N elements and saves the result.	3	1	1
scale	Multiplies N elements of an input stream by a scalar and saves the result.	4	6	6
sub	Subtracts one input stream from another element-wise over N elements and outputs the result.	4	1	1
main cos	Scales with frequency to obtain A (see Equation 4.1), applies cosine, and multiplies with real visibilities.	4	2	2
main sin	Scales with frequency to obtain A (see Equation 4.1), applies sine, and multiplies with imaginary visibilities.	4	2	2

Table 5.3: Summary of Kernel Functions of Implementation 2.

performance achieved by the same kernel without the LUT operations as seen in Figure 5.6 labeled ”main (w/o LUT)”, which achieves an execution time of less than one-ninth of the original.

5.1.4. Output Parsing

For all implementations using MLIR-AIE, the output comes as a one-dimensional array of bytes of size N * 2, where N is the product of the dimensions of the image, and this is times 2 because of the bfloat16 datatype. Thus, the output must then be cast to bfloat16 and reshaped/reinterpreted to match the dimensions of the image. Another feature is that the implementations experience issues with input and output of NaN values, which are present at the edges of the image projection. In this case, those pixels come out of the NPU as 0 or very large values, depending on the implementation. Given that these are not relevant, they can be discarded or, for visualization purposes, they can be substituted with NaN values using a nanmask as in Listing 5.3.

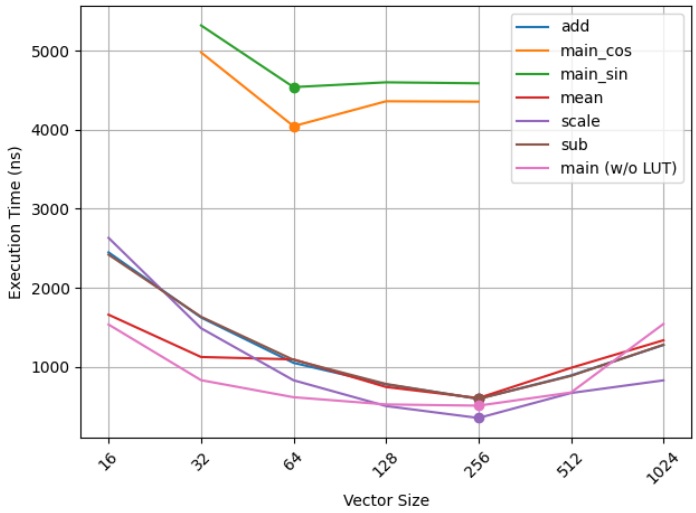


Figure 5.6: Execution time of the kernels from Implementation 2 given by the vector size used in the program.

```
bufOut = bo_inout4.map<bfloat16 *>(); // casting the output to bfloat16
for(auto i=0; i<OUT_SIZE; i++)
    if(nan_mask_v[i])
        bufOut[i] = static_cast<bfloat16>(nan("0")); // insering the NaN for
            visualization
```

Listing 5.3: NPU output formatting

5.1.5. Tracing

For each of the MLIR-AIE implementations, tracing was enabled as an optional feature to run with the application. To enable tracing, the application is called with an additional buffer object for storing the tracing output. This comes in the shape of an int8 array, similar to the regular application output.

The output is then cast to int32 and parsed using the *parse_trace* function, which is provided via the MLIR-AIE repository. Ultimately, the tracing output has to be fed to a tracing visualization software to see the event timeline. The recommended program by the development team is called Perfetto [67], which is a web-based trace visualization tool.

Despite the usefulness of tracing for most applications, in this case, the tracing did not provide useful information for performance and overall implementation behavior. This is because of the limited number of events that are recorded by the hardware. This, in turn, does not give a good idea of the overall execution time of the application, and it is not clear whether the recorded events show the full cycle count of the application or not. Nevertheless, there could be some uses for the tracing feature, for instance, to see the order of specific events or to make sure some things happen in a certain order in the array. Therefore, for the rest of the implementation evaluation, this feature is not used.

5.1.6. HDF5 Input Parsing

HDF5 (Hierarchical Data Format version 5) is a file format designed for storing and managing large and complex datasets. Developed by the HDF Group, it supports the storage of multidimensional arrays, images, tables, and metadata in a structured, efficient, and scalable way. This structure also supports compression and parallel I/O, which makes it particularly well-suited for high-performance computing environments and applications [68].

HDF5 is widely used across a variety of fields that require large amounts of data processing. It is commonly used in fields such as Physics and Astronomy as a standard for storing simulation results, telescope imagery, and particle data. Its flexibility and support across multiple programming environments have made it a preferred choice in many other domains requiring robust data management [68].

An important feature of the imaging application is to use the most common file formats, where the input data is saved for testing and analysis purposes. As with many other scientific endeavors, the

HDF5 format is commonly used to save signal data from the stations. However, the data is not usable directly for the NPU implementations described in this report, so it needs to be parsed from the files and processed to meet the correct format. For this project, the use of HDF5 files was essential for obtaining real-life data and assessing the accuracy of the implementation.

The structure of antenna sample data in the HDF5 file is structured as a group containing metadata about the datasets, and under it, a series of datasets, each containing a set of visibilities. The number of datasets in the file is the same as the number of subbands the antennas can detect. A file containing LBA data, for instance, contains 512 datasets, each with a distinct sampling frequency, which covers the entire sampling range of the antennas. The group also contains metadata about the datasets called attributes. Two relevant attributes for this implementation are the ”antenna_reference_itrf”, from which the baselines are calculated, and the ”subband_frequencies”, which contains a list of the subband frequencies of the datasets in order. The main file group is usually named ”/” because it serves as the source of the file.

Parsing

The objective is to parse the data contained in the HDF5 file to obtain the inputs of the algorithm, which are: visibilities, baselines, and frequency. The cosine directions can be generated without the need to extract them from a file.

The baselines are obtained from the antenna_reference_itrf attribute in the XYZ-coordinate system. This needs to be transformed to be aligned with the antenna array; otherwise, the image generated appears to be blurred. This is done by multiplying the baselines' matrix by the rotation matrix and then getting the PQR coordinate system.

To obtain the frequency and visibilities, each of the datasets in the file must be parsed. Each of the datasets stores the data for the subband index in its name. The frequency can be obtained by using the index in the file name with the ”subband_frequencies” attribute. The visibilities are extracted from the dataset, and then the real and imaginary components need to be extracted since the data is stored twice, once with the required data and its complex complement.

5.2. Implemtation 3: Ryzen AI Software (TINA)

The implementation of Solution 3 was done using PyTorch because of the ease of use and widely available documentation, and because it is the only tool currently supported with TINA. Because of issues with quantization during development, the implementation was made with float32 as the data type of the inputs. The development pipeline was done in three steps:

1. Model Definition
2. Exporting Model
3. Running the Model

The model implementation consists of a PyTorch NN module that takes the visibilities input as an argument for the *forward* function. Listing A.2 shows the implementation for the main layer model called *all_sky_image*. This layer is for model initialization and calls another PyTorch NN module containing the convolution called *inner_conv* and is shown in Listing A.3. Since there is no training involved in this model, the weights for *inner_conv* are the values of the exponential matrix, calculated with the function in Listing A.3.

The *inner_conv* utilizes the *nn.Conv2d* as the model's convolution. The convolution has 2 channels for the real and imaginary values, a batch size of $(npix\_l, npix\_m)$ for calculating each pixel, a kernel size of 96^2 representing the size of the baselines/visibilities, and is set with a stride size of $(1, 1)$. This convolution is then set as a transformation of $[2, npix\_l, npix\_m, 9216] \Rightarrow [2, npix\_l, npix\_m]$.

The implementation of the exponential matrix follows the definition in Equation 4.9. This function is called in the layer initialization function of *inner_conv*, by taking the baselines, frequency, and output image dimensions as arguments. The input size for the exponential matrix function is $InputBytes = (1 + 3 * 9216 + 2) * DataTypeBytes$, which $DataTypeBytes = 4$ because of the float32 type. The input size for the implementation is given only by the size of the visibilities $InputBytes = 2 * 9216 * DataTypeBytes$, greatly reducing the input data size when compared to the other implementations

The model build process can be seen in Listing 5.4, where the *all_sky_image* layer is called with the input data for the baselines, frequency, and output image dimensions, all formatted in a similar way to the other implementations. After the input and output shapes are defined to export the model ONNX. Once this is done, the model can be run with the visibilities in the correct tensor format.

Listing 5.4: Building and exporting the PyTorch model to ONNX

```
# Instantiate the model
pytorch_model = all_sky_image(baselines, frequency, npix_l, npix_m)
pytorch_model.eval()

# Export ONNX
input1 = torch.randn(1, 2, 96, 96) # input shape of the visibilities
inputs = {"x": input1}
dynamic_axes = {"input": {0: "batch_size"}, "output": {0: "batch_size"}}
model_path = "models/lofty.onnx" # ONNX model file location
torch.onnx.export(
      pytorch_model,
      inputs,
      model_path,
      export_params=True,
      opset_version=17, # Recommended opset
      input_names=['input'],
      output_names=['output'],
      dynamic_axes=dynamic_axes,
   )
```

For running the implementation visibilities values have to be in the shape of $[2, 96, 96]$, the real and imaginary components concatenated in that order. The input formatting can be seen in Listing 5.5, where the exponential matrix is obtained, separated by real and imaginary, and then concatenated into one 3d tensor. The model is run using an ONNX Runtime session, and then the inference function is called, as shown in Listing 5.6.

Listing 5.5: Running the ONNX model in the NPU

```
# Compile
install_dir = os.environ['RYZEN_AI_INSTALLATION_PATH']
config_file_path = os.path.join(install_dir, 'voe-4.0-win_amd64', 'vaip_config.
    json') # Path to the NPU config file

aie_options = onnxruntime.SessionOptions()

aie_session = onnxruntime.InferenceSession(
   model.SerializeToString(),
   providers=['VitisAIExecutionProvider'],
   sess_options=aie_options,
   provider_options = [{'config_file': config_file_path,
                  'cacheDir': cache_directory,
                  'cacheKey': 'tina_cache'}]
)
npu_results = aie_session.run(None, input_data) # Running the model
```

Listing 5.6: Input formatting for Implementation 3

```
# Separating visibilities
vis_tensor_real = torch.from_numpy(np.real(visibilities)).float()
vis_tensor_imag = torch.from_numpy(np.imag(visibilities)).float()
shape_input = vis_tensor_real.shape

vis_tensor_real = vis_tensor_real.view(1, 1, shape_input[0], shape_input[1])
vis_tensor_imag = vis_tensor_imag.view(1, 1, shape_input[0], shape_input[1])
vis_tensor_comp = torch.concat((vis_tensor_real, vis_tensor_imag), dim=1)
```

5.2.1. Operating System Compatibility & Model Size

The current version of Ryzen AI SW only supports Windows OS to run ONNX models. Because Phoenix1 is a Linux machine, an alternative environment had to be acquired to run the Ryzen AI SW application. To solve this issue, AMD provided remote access to a Phoenix PC with Windows.

Because of the model size limit of the ONNX exporting tool of 2 GiB, the current implementation does not allow some of the common image sizes like 128x128. Thus, the largest image generated with Solution 3 is a 100x100 image. Regardless, this is enough to measure the performance of the implementation and the scaling prediction for larger images.

6

Results

This chapter presents the numeric and nominal results of the 4 NPU implementations of the All-Sky Imaging Algorithm. The results are also justified using previous predictions and compared to the baseline implementations. Most of the results are obtained using 128x128 image dimensions since these are the most commonly used in practice. Ultimately, the results are compared and evaluated to determine if the implementation meets the requirements set at the beginning of the report.

6.1. The Baselines

This section provides an overview of the baseline implementations of the All-Sky Imaging algorithm using both CPU and GPU. The baselines will serve as a reference for evaluating the performance metrics of the NPU implementations. By comparing the results obtained to these baselines, the aim is to showcase the strengths and limitations of the new implementations and whether they are a good alternative in the field.

There are 2 types of implementations used to establish a baseline performance for this thesis: CPU and GPU. There are 2 CPU implementations, one implemented using C++ vectors and the other one using NumPy (see Listing 2.1). On the other hand, GPU implementations are used to compare the performance of the NPU to other accelerator hardware. The GPU implementations were provided by the ASTRON team and are part of a repository called 'lofty' [69], where multiple implementations of the All-Sky Algorithm and example data are stored.

The C++ implementation was used for getting the accuracy results for the MLIR-AIE due to the better compatibility with the HDF5 library. This implementation has a very similar structure to the NumPy version, but uses the modified algorithm with sine and cosine instead of the exponential, and switches the NumPy arrays for standard library vectors. Listing A.1 shows the code for the C++ implementation used in this project. Table 6.1 shows the performance results of the baseline CPU implementations recorded in the Phoenix1 CPU. Section 6.4 will expand on how these results were obtained.

Version	Dimensions	Exec Time (s)	Avg Power (W)	Total Energy (J)
C++ vector	128x128	2.3507	29.0859	53.8511
NumPy	128x128	1.8514	29.4972	69.3391

Table 6.1: Performance results of CPU implementations of the All-Sky Imaging Algorithm run in the AMD Ryzen 7 8700G.

The GPU implementations were developed using JAX. JAX is a library for array-oriented numerical computation with automatic differentiation and JIT compilation for high-performance applications [64]. JAX also enables the applications to run in accelerator hardware like GPUs and TPUs. However, in order to use JAX, the target GPU must be compatible with CUDA, and unfortunately, the integrated GPU in the Phoenix is not compatible. Regardless, these results show a good point of comparison to alternative accelerated implementations, even though they are not a good alternative for the NPU implementations given these circumstances.

Table 6.2 shows the execution time and power for GPU implementations running the All-Sky Algorithm using the Intel 155H processor, which has an Intel Arc iGPU. In terms of performance, the Intel

Arc iGPU is less powerful in comparison to the AMD Radeon 780M of the Phoenix, with 33% fewer graphics cores and a 22.41% slower clock frequency. Thus, if the Phoenix iGPU was compatible with CUDA/JAX, the expectation is that these implementations would be around 2.28 times faster. The results include a variety of versions of the JAX implementations. The versions labeled ”Real” are using the adapted version of the algorithm as seen in Equation 4.5. The versions labeled ”Ravel” use a flattened version of the algorithm. Finally, the version labeled ”Precompute” uses precomputed data of $\mathcal{A}$ as in Equation 4.1. The results of the GPU implementations show a great deal of acceleration and an increase in power consumption when compared to the CPU implementations, but because of the short execution time, the overall energy consumption is lower.

Version	Dimensions	Exec Time (s)	Power (W)	Energy (J)
Jax	128x128	0.068	86.4853	5.881
Jax Ravel	128x128	0.082	39.1098	3.207
Jax Ravel Real	128x128	0.073	45.3151	3.308
Jax Real	128x128	0.074	38.2162	2.828
Jax Precompute Real	128x128	0.014	43.7143	0.612

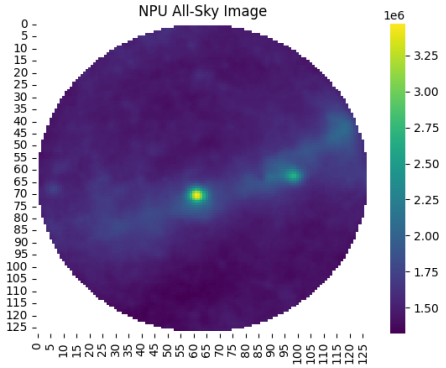
Table 6.2: Performance results of GPU implementations of the All-Sky Imaging Algorithm run in the Intel Arc Graphics iGPU.

6.2. Accuracy

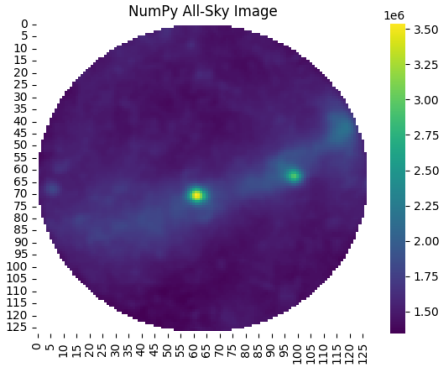
This evaluation aims to assess whether the change in hardware has any significant impact on the system’s accuracy to determine the usability of the new implementations. This is done by comparing the output from the NPU to the CPU baseline. To have an accurate evaluation of all the use cases of the All-Sky Imaging application, the output images for several subbands are used. In particular, the accuracy evaluation was done for the frequency subbands of the LBAs due to real input data availability. The expectation is that the resulting image frames give a good representation of what is being observed in the field. Also, the average error of the implementations must be within 5%. This was established as an acceptable margin of error for diagnostic images according to ASTRON instrument scientists.

6.2.1. The Results

The accuracy of the implementations can be determined in two ways. The first is the so-called ”eye test”; given that the application would be used for diagnostics, it is important that it is good enough for humans to use, regardless of what the numbers say. In this case, by visually comparing the images to their reference, they appear to be nearly identical, with only a small amount of difference in color hue. This can be seen where Figures 6.1(a) and 6.1(b) appear to have minimal difference, with the first being the output for Implementation 1, and the second being the output for the NumPy baseline implementation. Nevertheless, it is qualitatively accurate enough to pass this test.



((a)) Output image of Implementation 1 of the All-Sky Imaging algorithm.



((b)) Output image of NumPy All-Sky Imaging algorithm implementation.

Figure 6.1: Output images of Implementation 1 and the NumPy implementation of the All-Sky Imaging algorithm.

The second method to measure the accuracy of the output is by comparing the pixel values of images generated by the NPU applications with the reference. There are several different ways to go about this. The results from this method can be seen in Figure 6.2. These results are obtained by getting the ratio of the difference between the implementation and the reference, and the reference. Then, the error percentages for all the pixels of the image are averaged. This is done for the output of each subband.

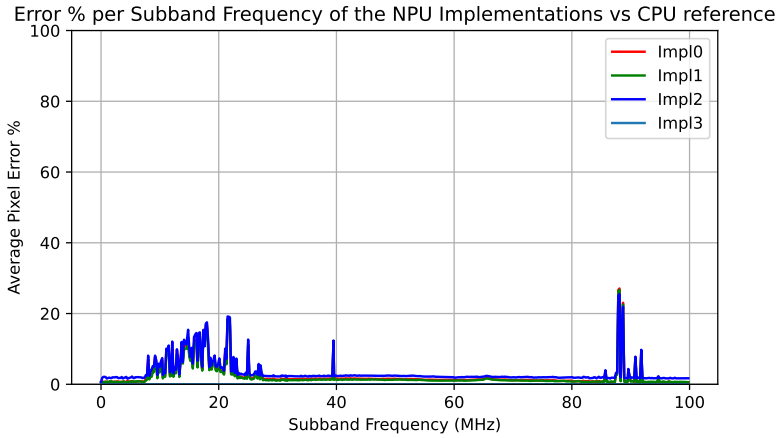


Figure 6.2: Error % of the NPU Implementations by subband in comparison to CPU reference

Impl	Mean Avg Error %	Max Avg Error %
0	2.3852	27.0848
1	2.1686	26.6932
2	3.2782	25.4806
3	0.0001	0.0004

Table 6.3: Error % metrics for the NPU implementations.

In Table 6.3, for the MLIR-AIE implementations, the accuracy for all three implementations has a very similar distribution, with a very similar average and max error percentage. The results were made using the standard image size of 128x128. However, these numbers still have some variation between them despite the fact that they are all doing the same calculation. The most likely explanation is that because the implementations use different amounts of inter-tile communication in the NPU, the difference in the number of conversions between accumulators, which use float32, and vectors may cause a small amount of inaccuracy. This would explain why Implementation 1 has the lowest error percentage, followed by Implementation 0, and then finally Implementation 2.

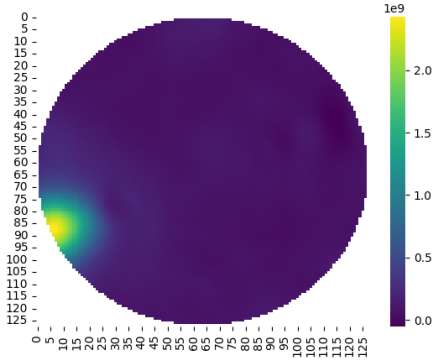
The accuracy for Implementation 3 is much better than the other implementations. Given the current limit on the image size, the results for the TINA implementation were made using a 100x100 image size in order to try to approximate the image size for the other implementations. Nevertheless, this level of accuracy can be explained by the fact that the implementation uses float32 instead of bfloat16 like the others.

6.2.2. Causes for inaccuracy

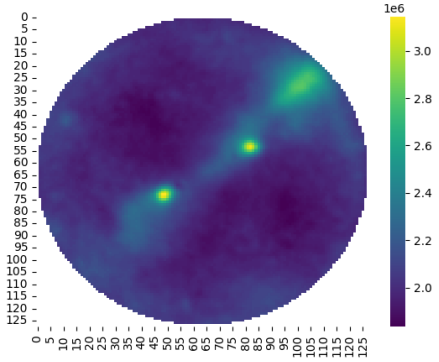
Multiple reasons can cause inaccuracies when it comes to reconstructing sky images with aperture synthesis. For this implementation, three reasons were identified that can explain the inaccuracy of the results:

- Lack of antenna sensitivity, most prevalent at both ends of the subband range.
- The presence of FM radio interference, also most prevalent at the ends of the subband range.
- The data type used in the NPU implementations (bfloat16 vs float32).

The first two cause the image to appear to have fewer features, with a single feature at the edge of the image, this being the interference source. An example of such an image can be seen in Figure 6.3(a), where the image does not show a clear shape in comparison to a less noisy image, such as Figure 6.3(b). Moreover, the error per subband in comparison to the reference appears to correlate with the RFI noise present at said subband. This phenomenon can be observed in Figure 2.7(a) and Figure 6.2. It is clear that the higher the noise, the more susceptible to error the imaging application becomes. Also, this serves as reassurance that the subbands with the highest error percentage require the least amount of precision, given their level of noise.



((a)) Noisy image sampled from subband 76 (16.8 MHz)



((b)) Noiseless image sampled from subband 286 (55.9 MHz)

Figure 6.3: Output images of the Implementation 1 of the All-Sky Imaging Algorithm

Also, the implementations that use bfloat16 appear to have an increased rate of error in these subbands with high RFI. This can be seen with Implementation 3 having a significantly lower error rate while using float32 instead of bfloat16. The error correlation with the type could be a combination of the precision in the intermediate results of the implementations, where adding values with a very different magnitude causes the loss of precision; this would accumulate over the computation of the pixel output and cause an error. The correlation with the RFI noise may be because the noise increases the signal's intensity. This increases the value range of the operators, thus causing the imprecision due to the bfloat16 data type's lower number of mantissa bits. Nevertheless, the accuracy for all implementations is within the margin that allows the applications to achieve their intended purpose. Added to that, the quality of the output images is mainly measured through visual inspection, so as long as this is met, the numerical evaluation is secondary.

6.3. Performance

The performance of the application is one of the main driving factors for this. The method for determining the performance of the different implementations is given by comparing their execution time to two factors: the performance of the baseline implementations and the potential performance given the execution platform and development tools. In the case of the MLIR-AIE implementations, the kernels running on the individual CTs are also evaluated using these methods.

To calculate the speed-up of the applications, these are compared to the Python baseline, as the main point of comparison, but also some of the other baseline implementations, to further reflect on the achieved results. For evaluating the achieved performance compared to the potential performance, the applications are graphed using the roofline model, where the rooflines are given by var0.4052ious factors regarding the hardware and the design choices of the applications. The applications are graphed using different image sizes as well as variations to the implementation to better explain the performance results and scalability. Also, the performance of the different implementations is compared to predictions made given their solution designs, and how these predictions can or cannot accurately portray the performance of similar implementations.

6.3.1. Overall

The implementations using the NPU appear to have achieved a great deal of speedup compared to the baselines. As seen in Table 6.4, all the NPU implementations achieved a speedup of at least 10 when compared to the NumPy baseline. This achieves the objective of processing the incoming data of the station with the highest throughput of 10 Hz.

Version	Image Dims	Exec Time (us)	NumPy Exec Time (us)	SpeedUp
Impl0	128x128	167837	1851564.04	11.03191811
Impl1	128x128	23925.3	1851564.04	77.3893761
Impl2	128x128	91812.4	1851564.04	20.16681886
Impl3	100x100	28249.1	1135680.68	40.2023668

Table 6.4: Performance speedup of the NPU implementations in comparison to the NumPy baseline.

Out of all of the NPU implementations, the fastest was Implementation 1, with a speedup of 77.4 compared to the NumPy baseline. The reasoning behind this can be explained because of the better distribution of the operations over the tiles used in the design, especially the LUT operations, which, as seen in the next section, use a large fraction of the total execution time. Also, it is the only implementation that takes advantage of skipping operations that would result in a complex output. The implementation with the second-best speedup results is Implementation 3. This level of performance can be algorithm reduction of the algorithm and optimization of TINA only using convolution operations. Then Implementation 2 and 0 take positions 3 and 4 in performance, both having a very similar implementation. As predicted in Chapter 4, the speedup of Implementation 2 is almost double that of Implementation 0 due to the split of the pipeline into two channels.

The NPU implementations also showed the predicted scalability as the number of pixels increased. Figure 6.4 shows the execution time of the NPU implementations using different numbers of pixels; the labels on the x-axis show N for an image of NxN pixels. Here it can be seen that the execution time grows linearly with the number of NxN pixels. Optimization to improve scalability would prove difficult since the calculation of the pixels is largely done independently from one another, and improving the linear relationship to pixels would require the reuse of calculations between pixels as the image grows in size.

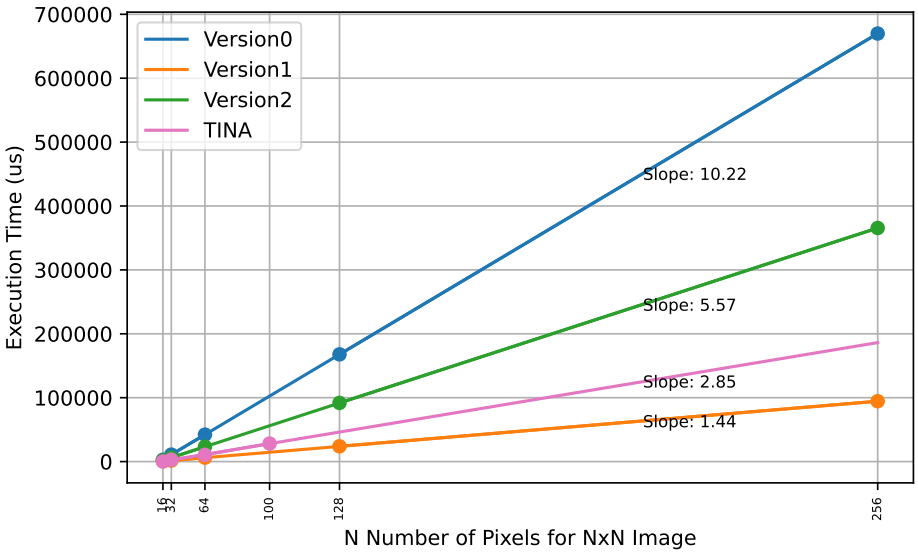


Figure 6.4: Execution time of the NPU implementations of the All-Sky Imaging Algorithm by the size of the output image.

All the implementations show a linear scaling relation, although the TINA implementation shows a weaker linearity compared to the rest. Another thing to point out is the smaller range for the TINA data-point sample due to the limit on the model size discussed in 5.2. Regardless, the model design does support this scalability model, given that the work increases linearly with the number of pixels of the output image.

Table 6.5 shows the speed of the NPU implementations compared to the fastest non-precompute GPU implementation, labeled ”Jax”. These results show that only Implementations 1 and 3 have achieved a speedup compared to the Jax version, with Implementation 1 getting a 2.84 speedup and Implementation 3 getting a 2.41 speedup. The results with Implementation 3 vs the Jax version do not hold much weight since they are using different image dimensions, so the Jax version has an unfair disadvantage in this case. If the execution time of Implementation 3 was extrapolated using the linear projection of Figure 6.4, the theoretical execution time for a 128x128 is 0.0464 s, thus having a theoretical speedup of 1.4657 compared to the Jax implementation. For Implementation 1, the results would also result in a speedup when using the theoretical execution time of the Jax version running on the Phoenix iGPU, which is 0.029 s, thus giving Implementation 1 a theoretical speedup of 1.24 compared to this version. These theoretical results require confirmation with experiments using equivalent hardware platforms, but they serve to give an idea of the achieved performance of the NPU implementations vs the GPU implementations.

Version	Image Size	Exec Time (s)	Jax Exec Time (s)	Speedup
0	128x128	0.1678	0.068	0.4052
1	128x128	0.0239	0.068	2.8451
2	128x128	0.0918	0.068	0.7407
3	100x100	0.0282	0.068 (!)	2.4113 (!)

Table 6.5: Speedup of NPU implementations running in AMD Ryzen NPU vs Jax implementation running in the Intel 155H iGPU.

6.3.2. Roofline Graphs & Achieved Performance Full Applications

Figure 6.5 shows the NPU implementations plotted in terms of their Arithmetic Intensity vs Performance, and the rooflines represent different levels of resource usage and assumptions given the potential performance of the applications in this platform.

The MLIR-AIE implementations are labeled as *vn_N*, where *n* is the version number for the implementation, and *N* is the number of pixels for a *N*x*N* image. For Implementation 3, the plot label is *TINA_N* where N is the number of pixels for a *N*x*N* image. Each implementation is plotted twice with different image sizes to observe the impact of the output image size on the relation between Performance vs Arithmetic Intensity. Also, for the MLIR-AIE implementations, the graph plots them again but removes the LUT operations from the execution, which correspond to the data points using a cross, and adds to the label ”(w/o LUT)”. This is to reflect on the impact of the LUT operations on the performance.

The arithmetic intensity is calculated by taking $ArithIntensity = Work/InputBytes$, where $Work$ is the number of bfloat16 operations (BFL16OPs) done during the execution of the implementation, while the $InputBytes$ is the total input size in bytes calculated in Chapter 5. The performance was calculated with $Performance = Work/ExecTime$, using the same $Work$ as before, and the $ExecTime$ is the execution time of the implementation. For Implementation 3, using TINA, the implementation still uses float32 as its working data type, rather than bfloat16. To give a fair assessment of the implementation, the calculation of the $Performance$ is doubled for Implementation 3, given the difference in performance for float32 vs bfloat16.

A key point to note is that there are two types of rooflines in Figure 6.5. The ones labeled as ”Naive” use the peak performance for the Compute Tiles given by the documentation. This peak performance is 0.8 Tera Operations per second (TOPS) or 800 Giga Operations per second for this graph (GOPS). This number does not specify the type of operations that are considered; these have to be taken with a grain of salt at the moment of comparing with real implementations, since this could be for a much smaller integer type. For the sake of the evaluation, this will be considered all to be bfloat16 operations, given that there is no clarification present, but this is likely not the case.

On the other hand, the rooflines labeled as ”Measured” use the performance measured with a very large and simple implementation in an attempt to calculate the performance for the main operations available in the hardware, similar to GEMM [70]. Table 6.6 shows the performance for the operations add, subtract, multiply, and multiply-and-add (MAC). Here, the measurements were made using a wide range of vector sizes, which shows the peak performance achieved among them per operation. The

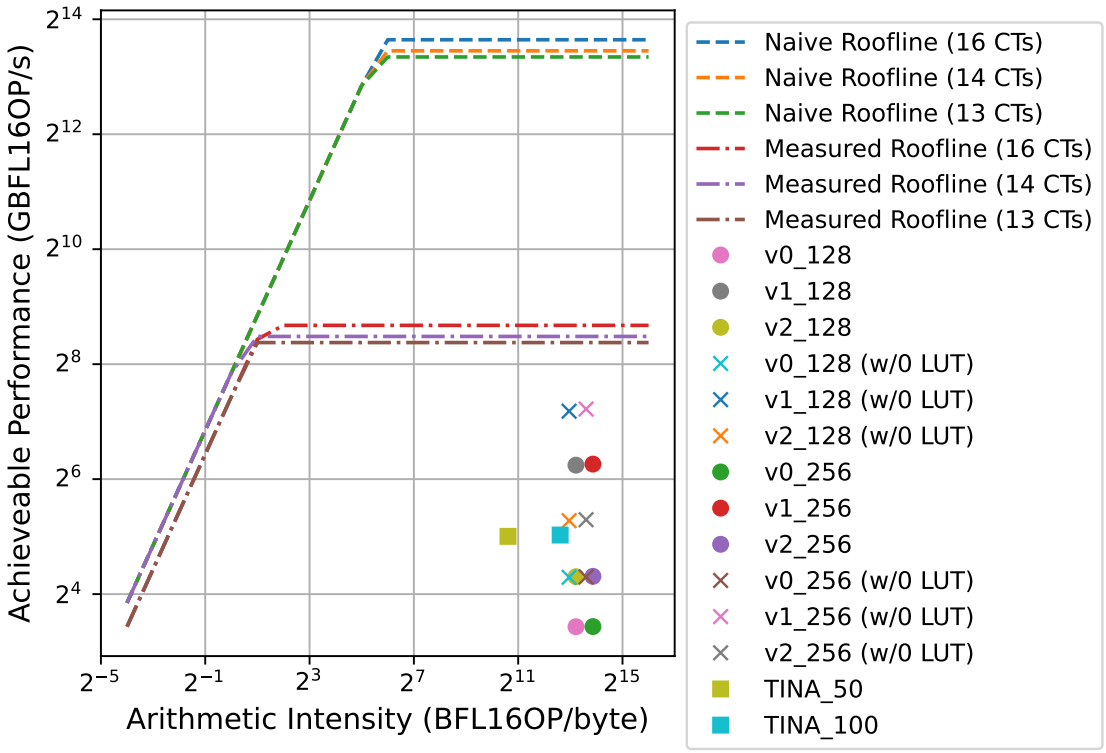


Figure 6.5: Roofline model graph of the NPU implementations given the naive and measured peak performance metrics of the AIE-ML.

results show a much lower peak performance, this being 25.55 GOPS of bfloat16 doing multiplication using a vector of 128 elements. Using this as the assumed peak performance, the implementations show to be much closer to achieving the limit for the current hardware. It is not clear what the cause of this disparity is; however, the answer falls outside of the scope of this thesis.

Moreover, for each roofline type, three rooflines are present in the graph. Each is labeled with the number of CTs that are assumed to be used to achieve this performance. This reflects the number of CTs used by the different NPU implementations, with Implementation 1 using 13 CTs, Implementation 0 using 14 CTs, and Implementations 2 and 3 using 16 CTs. This is to give a better estimation for the peak performance of each implementation given resource use, and also given the total available resources. With Implementation 3, it is unclear how many tiles are used by the Ryzen AI SW compiler due to its black-box nature, so it is assumed that all were used.

vec_size/op (us)	**add**	**sub**	**mul**	**mac**
w/o kernel	91.3	91.3	91.3	91.3
16	65421	65504.5	74686.1	65694.2
32	65610.5	65653.7	74996.3	65703.1
64	93627.4	93598.3	74969.1	93655.1
128	130895	130839	84041.3	130866
256	205392	205488	224109	205465
512	1.13E+06	1.13E+06	848602	979160
1024	2.37E+06	2.39E+06	2.13E+06	2.38E+06
best performance (GBFL16OPS)	20.91107393	20.90130468	**25.5527181**	20.9036444
vec_size for best perf	256	256	128	256

Table 6.6: The measured performance in GOPS for main operations used in IRON with the bfloat16 datatype.

As expected from the overall performance results, Implementation 1 found the most success in

achieving the highest performance among the NPU implementations. This is followed by Implementation 3, then Implementation 2, and finally, Implementation 0. This is the expected result because the work and input data size for each of the implementations is almost the same, thus being semi-aligned in the x-axis, and the difference in the y-axis is a reflection of the difference in performance.

For the MLIR-AIE implementations, Figure 6.5 also shows the difference in performance when removing the LUT operations. The versions without the LUT operations achieved, on average, more than twice the level of performance. This, in turn, explains the gap between the implementations and the peak measured performance of the NPU device.

C++ Kernels

The kernels are plotted similarly, with Figures 6.6, 6.7, and 6.8 showing the roofline graphs for the kernels of Implementations 0, 1, and 2, respectively. Again, the rooflines labeled ”Naive” use the advertised peak performance, while the rooflines labeled ”Measured” use the measured peak performance from Table 6.6. Also, the kernels that use LUT operations in their code are plotted without them to compare the achievable performance without them; these are labeled the same with ”(w/o LUT)” added at the end.

The rooflines are also classified according to the bandwidth. Given that the distance between tiles significantly affects the speed of data transfer, there must be a distinction for the peak bandwidth achievable by kernels. This is reflected in two versions of the ”Naive” and ”Measured” rooflines, the ones labeled ”High BW” which assume the incoming input streams are from neighboring tiles, while the rooflines labeled ”Low BW” assume that the incoming input streams a from non-neighboring tiles.

The roofline graphs for the kernels of the MLIR-AIE implementations show that most of the kernels are close to achieving peak measured performance. The exceptions are the kernels for ’sin’ and ’cos’ of Implementation 0 and the kernels ’main cos’ and ’main sin’ of Implementation 2. The common factor between these kernels is that they predominantly use LUT operations. The kernel ’main’ of Implementation 1 is an outlier in this sense because, despite using LUT operations, its performance is shown to be close to the roofline, like the other non-LUT kernels. When removing the LUTs from the kernel that uses them, their performance rises with the rest of the kernels; in particular, the ’main’ kernel of Implementation 1 is right below the measured peak performance.

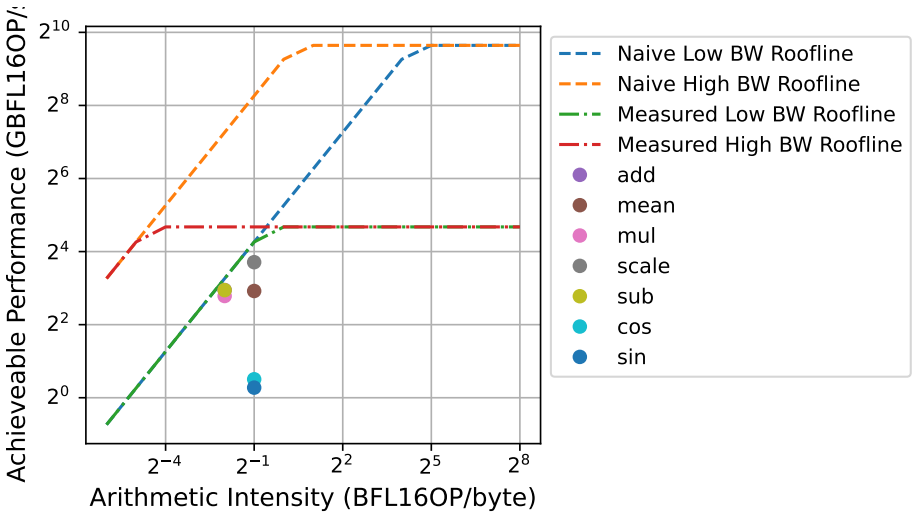


Figure 6.6: Roofline model graph of the C++ kernels of Implementation 0, given the naive and measured peak performance metrics of the AIE-ML.

6.4. Power & Energy Consumption

Part of the attractiveness of using the Ryzen AI devices is the claim that compatible applications can achieve performance comparable to using GPUs while maintaining a relatively low power consumption. Unfortunately, at the time of writing this thesis, there are a lot of limitations concerning measuring the power consumption of the NPU device. The XDNA driver can show the instantaneous power of the device for other models; however, this is not available with the Phoenix Model.

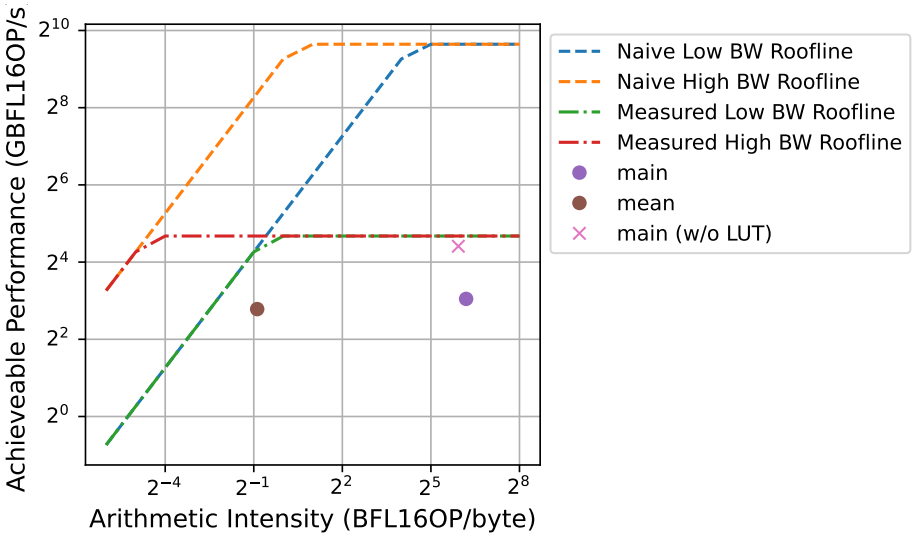


Figure 6.7: Roofline model graph of the C++ kernels of Implementation 1, given the naive and measured peak performance metrics of the AIE-ML.

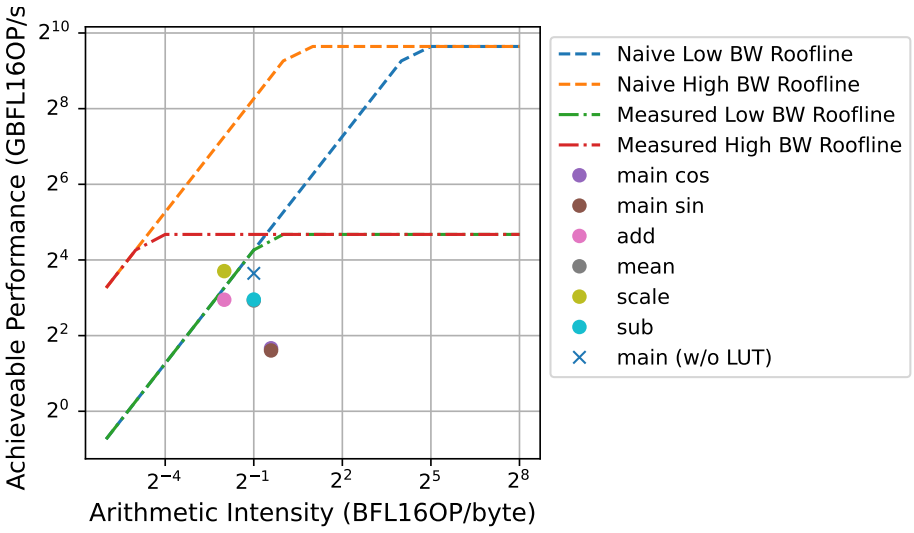


Figure 6.8: Roofline model graph of the C++ kernels of Implementation 2, given the naive and measured peak performance metrics of the AIE-ML.

Since Phoenix1 was installed with Linux, the system has access to the power capping framework. This framework exposes the power capping devices to user space via *sysfs*, which in turn is categorized with objects. At the top are the ”control types,” which correspond to different methods of power capping. The Zen4 processor has support for the ”intel-rapl” (Intel “Running Average Power Limit”) control type, which can report the energy and power of the processor by accessing the system registers. Under the control type exist the power zones. These represent specific devices that are within the processor, such as the CPU, iGPU, and, in this case, the NPU. However, most of the devices in the Zen4 processor are not supported for reporting with the powercap framework.

When going into the power zones of the powercap directory, the only devices that are listed are under ”intel-rapl:0” and ”intel-rapl:0:0”, and they have the names ”package-0” and ”core” respectively. By doing some preliminary measurements, and given that it is right below the intel-rapl control type, it is clear that ”package-0” corresponds to the CPU. On the other hand, it is not very clear what ”core” corresponds to, but given its power zone ranking and its name, it can be assumed that it corresponds to one of the CPU cores, probably core index 0, where sequential applications usually run. For these two devices and the control type as a whole, multiple sub-directories correspond to different metrics,

the most relevant of which are power and energy. Unfortunately, the power metrics appear as ”unsupported”, so only energy can be obtained from them, except for the top-level control type (”intel-rapl”), which supports neither. This means that only the energy values of the CPU can be accessed, thus not allowing the power measurements of the NPU.

Another gap in the power results is given by the platform where the TINA application was run. The remote server used to run the application did not have access to the terminal; thus, the power registers and other command-line capabilities were not accessible for this platform. In this case, running the application on a local machine and a model that supports more extensive metrics may lead to getting these results. Nevertheless, due to the time constraints of the project, this was not possible.

6.4.1. CPU Power Metrics

The energy registers for the CPU and the core can be read via ”/sys/class/powercap/intel-rapl:0/energy_uj” and ”/sys/class/powercap/intel-rapl:0:0/energy_uj” respectively. Reading these registers returns the energy in Joules consumed by these power zones since the last reset. It is not clear when this reset happens, but it is not necessary to know this to get the desired results. With this data, it is possible to obtain the average power consumption of the device by applying $Power = (Energy\_After - Energy\_Before)/Duration$. With this, it is possible to obtain the power consumption for the NPU implementations compared to the baselines and confirm the reduction in power consumption of the CPU.

Table 6.7 shows the average power consumption of the power of the available power zones for the different implementations. The baseline power consumption is given by two different CPU implementations, Listing 2.1 and A.1 using Python and C++, respectively. Also, it shows the idle power consumption in the first row. Finally, the three NPU implementations can be observed to consume an equivalent amount of power when compared to the idle CPU. This confirms that, despite not having the measurements of the NPU, it is clear that utilizing it allows for the freeing of CPU resources. Additionally, given the reduction in execution time when compared to the baselines, the total energy consumed by the application is reduced regardless.

App Version	Iterations	Total Time (s)	CPU Power (W)	Core Power (W)	E per Iter (J)
Idle	100	1	10.6413	0.00517693	1.06413
NumPy Baseline	100	185.1449	29.0859	0.051421	53.8511
C++ Baseline	100	235.07	29.4972	0.043662	69.3391
Impl0	100	16.7859	13.6583	0.0823	3.0815
Impl1	100	2.39814	11.1440	0.01611	0.2447
Impl2	100	9.19002	10.5292	0.0291	1.0597

Table 6.7: Power and Energy consumptions for All-Sky Imaging implementations for 128x128 image using CPU and NPU [NPU power not measured]

6.5. Solution Evaluation

This section presents a comparison of the 4 NPU implementations presented in this report. The comparison is in terms of the achieved performance and the overall usability as a tool to be used in the field. Table 6.8 evaluates their points in comparison and nominally grades these points. This serves as a summary for evaluating the fitness of the solution to the research question. The result summary shows that all the solutions have strong and weak points, but overall achieved good results in terms of performance and accuracy. However, there are some clear areas of improvement for all of them.

Version	Speedup	Versatility	Accuracy	Real-time	Potential Impr	Impl Difficulty
0	11.0319	High	Acceptable	No	Low	Medium
1	77.3894	High	Acceptable	Yes	High	Medium
2	20.1668	High	Acceptable	Yes	Medium	Medium
3	40.2024	Low	Excellent	Yes	High	Low

Table 6.8: Evaluation of the implementations of the All-Sky Imaging Algorithm for the AMD Ryzen NPU

6.5.1. Speedup & Performance

In terms of speedup compared to the baseline implementation, all implementations improved by at least one order of magnitude. This shows the potential of the AMD NPU hardware for DSP applications. However, not all of the implementations achieved the goal of computing at a rate of 10 Hz as established in Chapter 1, with Implementation 0 being the only one not reaching this goal for the standard image size of 128x128.

The results do show a clear preference for parallel designs rather than pipeline designs. With the three MLIR-AIE solutions, it can be seen that the performance is correlated with a more parallel design, with Implementation 1 being the fastest, then Implementation 2, and Implementation 0 being the slowest of the three. But this does not tell the whole story. A better explanation for these results is the parallelism of the LUT operations, which, as seen in Section 6.3.2, take most of the execution time of the implementations. Table 6.9 shows the speed up of the implementation weighted with the number of tiles computing the LUT operations.

Version	Speedup	LUT Tiles	Speedup/LUT Tiles
0	11.0319	2	5.5
1	77.3894 (60.7814)	12	6.4 (5.06)
2	20.1668	4	5.04

Table 6.9: Speed up of the MLIR-AIE implementations weighted with the number of tiles computing LUT operations.

This results in a much better comparison and explains the reason why there is such a disparity for implementations performing the same number of operations in the same hardware. The speedup of Implementation 1 also shows the weighted speedup when no operations are skipped in order to have a better comparison to the other implementations. When all of this is taken into account, there is a speedup of approximately 5 times faster for each tile that shares the load of the LUT operations with their current implementation.

The results of Table 6.9 also give an idea of the potential performance increase for this solution when implemented for an AMD NPU with more CTs, where Solution 1 will benefit greatly when compared to the others since the design ideas of Implementations 0 and 2 cannot increase the parallelism of the LUT operations much further. With regards to Implementation 3, the LUT operations are not an issue because these operations are skipped.

When compared to the GPU implementations, the performance of the NPU implementations shows a lower speedup than with the CPU implementations. In this case, the only NPU implementations that have a better execution time than all the GPU implementations (not counting the precomputed version) are Implementations 1 and 3. Even when taking into account the hardware disparity, and granting the 2.28 times speedup to the GPU implementations explained in Section 6.1, Implementations 1 and 3 have a lower execution time. Nevertheless, further testing is needed to make this assertion.

6.5.2. Accuracy & Data Types

The results of Section 6.2 show that the working data type of the implementations significantly affects the accuracy. This is clear when Implementation 3, which is the only implementation using float32, obtained a near-perfect accuracy in comparison to the reference. In spite of the acceptable results of MLIR-AIE implementations for the current requirements, it is important to recognize that in some specific cases, these inaccuracies might not be desirable.

The solution is to switch the working data type of the MLIR-AIE implementations from bfloat16 to float32. Implementation 3 demonstrates that, even though the support for this type is not complete in the hardware, it is good enough to meet the accuracy demands of this implementation. However, there are two issues with switching data types.

The first issue in switching from bfloat16 to float32 is the cached data size. This is an issue in particular for Implementations 0 and 2, which have to cache a large amount of data in some of the CTs, and switching to float32 might cause it to exceed the data memory limit. For Implementation 1, however, there is a good chance that it is possible to do the switch due to the low utilization of the data memory of the CTs. The current size of the cached data in the main CTs is $(768 * 5 + 196) * 2 = 8072$ bytes or 8 KB. This means that it can be stored in a single memory bank of 16 KB. Thus, by changing to float32 and duplicating the size of the cached data, the rest can be saved in another memory bank.

The second issue with switching from bfloat16 to float32 is the reduced speedup. As mentioned in Section 6.3.2, the use of float32 approximately halves the performance of an implementation in the Ryzen NPU in comparison to using bfloat16. This means that switching to float32 might cause the implementation to no longer pass the 10 Hz requirement. For Implementation 0, this is not even possible with bfloat16, so switching makes the implementation even more inadequate in terms of performance. For Implementation 2, the current version is already at the limit for its execution time, so switching to float32 causes the execution time to go over the limit to be able to generate image frames at 10 Hz. Lastly, Implementation 1 is a good candidate to make the switch to float32, since it has a wide margin in its performance and might benefit from an increase in accuracy. In the opposite direction, switching Implementation 3 to use bfloat16 (via quantization) may cause it to experience reduced accuracy. This would need to be weighed against the potential performance increase and whether it is worth further increasing the speedup of its current version.

6.5.3. Implementation Versatility & Potential Improvements

The versatility of the implementation refers to the ability of the implementation to run input types without having to be recompiled. All solutions might have to experience recompilation in the field as they are currently implemented, in particular when wanting to switch to a different output image size.

However, Implementation 3 experiences low versatility in this regard because the model only works for a fixed image size and sampling frequency. Thus, when wanting to generate images for multiple frequency subbands, the current implementation of solution 3 would struggle to meet the performance requirements due to the time spent in recompilation. The way to solve this is by moving the computation of the exponential matrix to run time rather than compile time, but it would be at the detriment of the performance of the implementation.

Another potential limitation is that having a fixed number of 96 antennas is a restriction on the current NPU solutions. This is for the cases when fewer station antennas are sampled or some of them are out of service. Currently, this can be worked around by zeroing out the unused antennas. This solution outputs a correct image, but the magnitude would need to be corrected if needed. Also, the computation of the unused antenna inputs wastes execution time and resources, so a more stable solution might need to be considered.

7

Discussion & Conclusion

7.1. Discussion

7.1.1. Results, with some caveats

The obtained results reflect the achieved performance of the different NPU implementations and how these can be improved upon. However, there a some limitations to the obtained results, which require the proper context to have a complete picture of the obtained performance and accuracy. The solution for these limitations may come in the shape of more expansive tests or modifying the methodology for obtaining the said results.

The performance results are the most comprehensive of the obtained results, given the required analysis to maximize the performance of the implementations and the hard requirements from the LOFAR stations. However, there are still ways to improve these results and provide a more comprehensive understanding of the solutions. One of the main ways to better evaluate performance is by getting a better estimation of the peak performance for the bfloat16 and float32 in the AIE-ML. The numbers given in the documentation and used in Section 6.3.2, are not well explained and give an unrealistic peak performance. On the other hand, the measured peak performance results were obtained with as little bias as possible; however, the results' evaluation would benefit from official metrics of the peak performance for the used data types, or a more extensive performance study, including the measured performance of other types, such as float32.

With regards to Implementation 3, the most notable limitation is the model size limit. Solving this issue would greatly improve the obtained results for this implementation, give a better idea of the scalability of the performance, and have a better comparison to the other implementations. This issue gives a good reason for attempting to also quantize the model, since switching to bfloat16 would reduce the model size, thus giving the chance to increase the output image size along with overall performance.

The accuracy results also experience some limitations that leave some gaps in the solutions' evaluation. The biggest gap is in the lack of results for the HBA frequency subbands. These results were not obtained due to a lack of example data at the time of implementation. However, even the LBA results could benefit from a larger sample size. The low availability of example data can be solved by generating artificial sample data, thus completing the missing sample frequencies.

Finally, the biggest gap in the results is the energy and power results. This issue is caused by a lack of support for the current hardware drivers to access the power registers, so there is little that can be done to solve it at the moment. One alternative is to measure the system's power externally. However, this requires specialized hardware that has not yet been tested on the Phoenix. Having the power consumption results would allow a complete evaluation of the implementation's viability in the field, so it is a necessity if this is to be deployed.

7.1.2. Choosing a solution

As discussed in Section 6.5, all implementations achieve a significant improvement in terms of performance compared to the baselines and result in an acceptable level of accuracy. However, for deployment, only one of them would be chosen to obtain the best possible results and user experience. As they are now, the implementations still need to be developed into full-fledged applications for field

use; nevertheless, the current results give a good idea of the solution quality and future performance in practice.

First, among the MLIR-AIE implementations, there is a clear answer in terms of both performance and accuracy, which is Solution 1. The performance results show a close-to-peak performance, as well as easy scalability into bigger implementations if needed. In terms of accuracy, the results do not vary much among the implementations, but by a small margin, Implementation 1 has a better accuracy than the other two. Another advantage of choosing Solution 1 is the opportunity to switch to using float32 data type, which has the potential to further improve the implementation's accuracy while staying within the performance requirements.

Ultimately, it comes down to choosing between Solution 1 and Solution 3. The advantage of choosing Implementation 3 is the increased ease of use and a better accuracy-to-performance ratio in comparison to Implementation 1. However, this comes at the cost of versatility, which would hinder the performance in the field compared to Implementation 1. With all of these points in mind, and as the implementations are currently, the more complete solution is **Implementation 1**.

There are some caveats to this decision. If the shortcomings of Implementation 3 can be solved effectively when it comes to model size, and the model used is extended to use the complete All-Sky Imaging algorithm at run time, then the easier implementation environment, Python vs IRON, would make it a more desirable solution. This is true especially when it comes to understanding the implementation and potential future modifications. Reiterating a previous point, this decision also needs to be backed by power and energy performance results to make sure the energy consumption requirements are being met.

7.1.3. Areas of improvement & Future Work

There are several areas of improvement for the NPU implementation presented in this report. Largely, these have to do with the accuracy and versatility of the models. Also, there are some features that, if added to the implementations, would provide a better user experience for the people involved in supervising the stations. The most clear areas of improvement are, in order:

1. Setting up a solution to the power measuring issue. Being able to measure the power consumption of the system is an essential component of determining the viability of software applications.
2. Extending Implementation 3 to use bigger models and the full algorithm. This extension would put it on an equal footing for comparison with the other implementations and give it the necessary versatility to become a contender as the chosen solution.
3. Make Implementation 0 use float32 to test the accuracy and performance changes compared to Implementation 0.
4. For all solutions, dynamically be able to run the application with different numbers of sampled antennas. This would make the current application use more seamless and reduce execution time in case of using fewer than 96 antennas in a given station.
5. For all solutions, dynamically be able to run the application with different output image sizes. This would make the application use more seamless and reduce the need for recompilation.
6. Other imaging applications. The lessons learned in this project can be applied to other applications used by people in the field to evaluate the conditions of the LOFAR stations.

7.2. Conclusion

The objective of this thesis was to determine the acceleration potential of the AMD Ryzen NPU when used to implement DSP applications, in this case, the All-Sky Imaging Algorithm. It also aims to analyze the effects on performance and resource use when compared to baseline implementations. ASTRON's project to implement an accelerated version of their All-Sky Imaging application serves as a good proof of concept to test the capabilities of the AMD Ryzen NPU.

The All-Sky Imaging algorithm is used in the field of radio astronomy to generate images of the sky using radio signals coming from astronomical sources. The objective of the NPU accelerated implementation is to have a real-time implementation of the algorithm used for diagnostics of the LOFAR telescope stations, requiring 10 Hz of image frame generation to reach the maximum rate of input data

generation. This is in contrast with the current baseline CPU implementation, with an execution time of 1.8 seconds per frame.

The project requires a degree of image accuracy compared to the existing baseline reference to qualify its correctness. This is to make sure that the quality of the images generated by the new implementations is good enough for the visual analysis of station operators. Also, the implementations aim to minimize resource use in terms of electricity consumption.

7.2.1. Solutions

The implementation was done with 2 different software toolchains, MLIR-AIE and TINA, and they were used to implement 4 solutions. The toolchains have different programming paradigms and levels of solution customization, resulting in varying solution strategies and performance. MLIR-AIE was used for the implementation of 3 solutions, while TINA was used to implement 1 solution. The MLIR-AIE implementations were designed using different heuristics to map the All-Sky Imaging Algorithm onto the NPU tile array, and the kernels running on each of the tiles were optimized to get the most performance out of the hardware. The TINA implementation used PyTorch to implement the algorithm, refactored using NN layers as its mathematical building blocks. The TINA implementation uses ONNX and Ryzen AI SW to run the application on the NPU.

Implementation 0 was designed as a pipelined implementation, mapping the mathematical operations of the All-Sky Imaging Algorithm to the NPU tile array as directly as possible, strictly dividing the operation types into the pipeline stages. It is seen as a Naive solution given the simplicity of the concept. The objective of this solution is to take advantage of the dataflow architecture design of the NPU while attempting to evenly distribute the computations over the pipeline.

Implementation 1 was designed to parallelize the operations of the All-Sky Imaging Algorithm by evenly distributing them over the tiles of the NPU array. This solution's objective is to evenly distribute the algorithm's operation to avoid having bottlenecks in the data path. This implementation shows the most potential for speedup given the highest level of distribution of the LUT operation out of the MLIR-AIE implementations. Also, it shows the most potential for scaling the implementation to more powerful systems hosting more NPU tiles.

Implementation 2 was designed as a middle ground between Implementations 0 and 1 by having a pipelined design but parallelizing it over two channels. The objective of this solution is to balance the previous 2 strategies in an attempt to get the benefits of both and maximize performance. It also helps in drawing a trend in the type of MLIR-AIE implementations and determining the best-suited design type for the NPU hardware.

Lastly, Implementation 3 uses TINA to implement the All-Sky Imaging Algorithm by refactoring using PyTorch convolutions and running it in the NPU to maximize its design to accelerate AI applications. This solution attempts to leverage the advantages of TINA to accelerate the non-AI application in AI hardware. To map the algorithm to the TINA building blocks, the portion of the algorithm computed during run-time is reduced to precompute the trigonometric operations. This is because, unlike MLIR-AIE, TINA does not have an alternative, given that the hardware does not support non-linear operations.

7.2.2. Results

The performance achieved by the implementations varied widely, with at least 10 times speedup compared to the CPU baseline. Out of all the implementations, the one with the best overall speedup was Implementation 1, achieving a speedup of 77.4 compared to the NumPy baseline and 2.84 compared to the JAX (GPU-run) version. From these results, Implementation 0 is the only one that did not meet the execution time requirement to generate image frames at a rate of 10 Hz.

Regarding accuracy, all implementations achieved an acceptable level, making them suitable for use as a diagnostic tool for the LOFAR stations. The average error percentage for the MLIR-AIE implementations is approximately 3%, with Implementation 1 having the lowest error rate by a small margin. On the other hand, Implementation 3 has an error percentage close to 0. This can be attributed to the difference in data type, where the MLIR-AIE implementations use bfloat16 and TINA uses float32. This, paired with the RFI noise in some of the subbands, causes an increased error rate for the implementations using bfloat16 due to the lower precision.

Unfortunately, the power consumption results could not be obtained due to a lack of support for the NPU hardware. The reason is that the driver does not support the model to access the power registers of the system. The power results need to be obtained to have a complete analysis of the

implementation and have a more informed decision on whether to deploy one of the solutions.

Ultimately, considering all the results, the solution with the best fit for eventual deployment in the field is Implementation 1. This is due to its speed compared to other implementations and its versatility for various use cases. Moreover, there are some areas of improvement that may prove this solution more useful for the operators of the LOFAR stations.

7.2.3. Contributions

The main contributions of this work are:

- **Implementation of DSP application using AMD Ryzen NPU**: This thesis serves as a proof of concept for the implementation of other applications using the AMD Ryzen NPU hardware, and explores the potential performance gains that can be achieved.
- **Accelerated implementation of the All-Sky Imaging Algorithm**: The solutions serve as an accelerated alternative to the CPU baseline, meeting the basic requirements to become deployed in the field and free up CPU resources.
- **TINA Implementation for the AMD Ryzen NPU**: Implementation 3 becomes a new layer available for the TINA toolchain, further expanding its use cases and proving its advantages for a new hardware platform.

7.2.4. Limitations and Future Work

The possible future avenues for work:

1. Setup power measurement capabilities for the NPU device.
2. Expand the TINA implementations to support the full All-Sky Imaging Algorithm.
3. Expanding the functionality of the implementations to increase the ease of use, increase performance/accuracy, and make them more suitable for deployment.
4. Implement other LOFAR station applications leveraging the AMD Ryzen NPU.

Bibliography

[1] Sabrina M. Neuman, Brian Plancher, and Vijay Janapa Reddi. *The Magnificent Seven Challenges and Opportunities in Domain-Specific Accelerator Design for Autonomous Systems*. 2024. arXiv: 2407.17311.

[2] Biagio Peccerillo et al. “A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives”. In: *Journal of Systems Architecture* 129 (2022), p. 102561. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102561. URL: https://www.sciencedirect.com/science/article/pii/S1383762122001138.

[3] Jörg Henkel et al. “New trends in dark silicon”. In: *Proceedings of the 52nd Annual Design Automation Conference*. DAC ’15. San Francisco, California: Association for Computing Machinery, 2015. ISBN: 9781450335201. DOI: 10.1145/2744769.2747938. URL: https://doi.org/10.1145/2744769.2747938.

[4] Sabrina Neuman, Brian Plancher, and Vijay Janapa Reddi. “Invited: The Magnificent Seven Challenges and Opportunities in Domain-Specific Accelerator Design for Autonomous Systems”. In: *Proceedings of the 61st ACM/IEEE Design Automation Conference*. DAC ’24. San Francisco, CA, USA: Association for Computing Machinery, 2024. ISBN: 9798400706011. DOI: 10.1145/3649329.3663515. URL: https://doi.org/10.1145/3649329.3663515.

[5] William J. Dally, Yatish Turakhia, and Song Han. “Domain-specific hardware accelerators”. In: *Commun. ACM* 63.7 (June 2020), pp. 48–57. ISSN: 0001-0782. DOI: 10.1145/3361682. URL: https://doi.org/10.1145/3361682.

[6] IBM. *What are large language models (LLMs)?* Accessed in 2025. 2023. URL: https://www.ibm.com/think/topics/diffusion-models.

[7] Dave Bergmann and Cole Stryker. *What are diffusion models?* Accessed in 2025. 2024. URL: https://www.ibm.com/think/topics/diffusion-models.

[8] J. Bathia. *AI Accelerators: Tracing the Past, Understanding the Present, and Forecasting the Future*. 2023. URL: https://medium.com/@jaskaranbhatia/ai-accelerators-tracing-the-past-understanding-the-present-and-forecasting-the-future-8c7fea7b5252.

[9] Tamador Mohaidat and Kasem Khalil. “A Survey on Neural Network Hardware Accelerators”. In: *IEEE Transactions on Artificial Intelligence* 5.8 (2024), pp. 3801–3822. DOI: 10.1109/TAI.2024.3377147.

[10] B. Lee and A.R. Hurson. “Dataflow architectures and multithreading”. In: *Computer* 27.8 (1994), pp. 27–39. DOI: 10.1109/2.303620.

[11] J. Schneider and Smalley I. *What is a neural processing unit (NPU)?* Accessed in 2025. 2024. URL: https://www.ibm.com/think/topics/neural-processing-unit.

[12] Alejandro Rico et al. “AMD XDNA™ NPU in Ryzen™ AI Processors”. In: *IEEE Micro* (2024), pp. 1–10. DOI: 10.1109/MM.2024.3423692.

[13] *TINA: Running non NN algorithms on an AMD Ryzen NPU!* Accessed in 2025. URL: https://www.hackster.io/tina/tina-running-non-nn-algorithms-on-an-amd-ryzen-npu-0cc58c.

[14] Peter Barry and Patrick Crowley. “Chapter 11 - Digital Signal Processing Using General-Purpose Processors”. In: *Modern Embedded Computing*. Ed. by Peter Barry and Patrick Crowley. Boston: Morgan Kaufmann, 2012, pp. 317–346. ISBN: 978-0-12-391490-3. DOI: https://doi.org/10.1016/B978-0-12-391490-3.00011-4. URL: https://www.sciencedirect.com/science/article/pii/B9780123914903000114.

[15] M. P. van Haarlem et al. “LOFAR: The LOw-Frequency ARray”. In: *Astronomy & Astrophysics* 556 (July 2013), A2. ISSN: 1432-0746. DOI: 10.1051/0004-6361/201220873. URL: http://dx.doi.org/10.1051/0004-6361/201220873.

[16] Primaluce Lab. *Introduction to radio interferometry*. Accessed in 2025. 2025. URL: https://www.primalucelab.com/blog/introduction-to-radio-interferometry/.

[17] P. Chris Broekema et al. “Cobalt: A GPU-based correlator and beamformer for LOFAR”. In: *Astronomy and Computing* 23 (2018), pp. 180–192. ISSN: 2213-1337. DOI: https://doi.org/10.1016/j.ascom.2018.04.006. URL: https://www.sciencedirect.com/science/article/pii/S2213133717301439.

[18] Xilinx. *MLIR-AIE*. URL: https://github.com/Xilinx/mlir-aie.

[19] *The History of Radio Astronomy - National Radio Astronomy Observatory*. 2022. URL: https://public.nrao.edu/radio-astronomy/the-history-of-radio-astronomy/.

[20] James J. Condon and Scott M. Ransom. *An Introduction to Radio Astronomy - Essential Radio Astronomy*. 2018. URL: https://www.cv.nrao.edu/~sransom/web/Ch1.html.

[21] Bernard F. Burke, Francis Graham-Smith, and Peter N. Wilkinson. *An Introduction to Radio Astronomy*. 4th ed. Cambridge University Press, 2019.

[22] European Space Agency. *Planck and the cosmic microwave background*. Accessed in 2025. 2025. URL: https://www.esa.int/Science_Exploration/Space_Science/Planck/Planck_and_the_cosmic_microwave_background.

[23] Helmenstine A. *Atmospheric Windows*. Accessed in 2025. 2023. URL: https://sciencenotes.org/electromagnetic-spectrum-definition-and-explanation/.

[24] AIP. *The First Telescopes*. Accessed in 2025. 2025. URL: https://history.aip.org/exhibits/cosmology/tools/tools-first-telescopes.htm#:~:text=The%20telescope%20first%20appeared%20in,objects%20three%20or%20four%20times..

[25] NASA. *Telescopes 101*. Accessed in 2025. 2025. URL: https://science.nasa.gov/universe/telescopes-101/.

[26] Nick Strobel. *Radio Telescopes*. Accessed in 2025. 2025. URL: https://www.astronomynotes.com/telescop/s4.htm.

[27] Office of Astronomy for Education. *Glossary term: Angular Resolution*. Accessed in 2025. URL: https://astro4edu.org/resources/glossary/term/284/r410/.

[28] A. Richard Thompson, James M. Moran, and George W. Jr. Swenson. *Interferometry and Synthesis in Radio Astronomy*. 3rd ed. Astronomy and Astrophysics Library. Open Access under CC BY-NC 4.0 license. Cham: Springer, 2017. ISBN: 978-3-319-44431-4. DOI: 10.1007/978-3-319-44431-4. URL: https://link.springer.com/book/10.1007/978-3-319-44431-4.

[29] NWO. *NWO Institute ASTRON: Netherlands Institute for Radio Astronomy*. Accessed in 2025. URL: https://www.nwo-i.nl/en/nwo-institutes-organisation/nwo-institutes/astron/.

[30] ASTRON. *ASTRON-Mission & Vision*. Accessed in 2025. URL: https://www.astron.nl/about/mission-and-vision/.

[31] *Aurorae discovered on distant stars suggest hidden planets*. Accessed in 2025. 2021. URL: https://www.astron.nl/aurorae-discovered-on-distant-stars-suggest-hidden-planets/.

[32] ASTRON. *Square Kilometre Array*. Accessed in 2025. URL: https://www.astron.nl/telescopes/square-kilometre-array/.

[33] *ASTRON LOFAR*. Accessed: 2024-11-20. URL: https://www.astron.nl/telescopes/lofar/.

[34] Gang Zeng et al. “A High-Performance Genomic Accelerator for Accurate Sequence-to-Graph Alignment Using Dynamic Programming Algorithm”. In: *IEEE Transactions on Parallel and Distributed Systems* 35.2 (2024), pp. 237–249. DOI: 10.1109/TPDS.2023.3325137.

[35] Supermicro. *What Is an Application Specific Integrated Circuit (ASIC)?* Accessed in 2025. URL: https://www.supermicro.com/en/glossary/asic.

[36] Adam Zewe. *Explained: Generative AI's environmental impact*. Accessed in 2025. 2025. URL: https://news.mit.edu/2025/explained-generative-ai-environmental-impact-0117.

[37] iea.org. *AI is set to drive surging electricity demand from data centres while offering the potential to transform how the energy sector works*. Accessed in 2025. URL: https://www.iea.org/news/ai-is-set-to-drive-surging-electricity-demand-from-data-centres-while-offering-the-potential-to-transform-how-the-energy-sector-works.

[38] Chris Lattner. *What about the MLIR compiler infrastructure?* Accessed in 2025. 2025. URL: https://www.modular.com/blog/democratizing-ai-compute-part-8-what-about-the-mlir-compiler-infrastructure.

[39] Nvidia. *PyTorch*. Accessed in 2025. URL: https://www.nvidia.com/en-us/glossary/pytorch/.

[40] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. "Advances in dataflow programming languages". In: *ACM Comput. Surv.* 36.1 (Mar. 2004), pp. 1–34. ISSN: 0360-0300. DOI: 10.1145/1013208.1013209. URL: https://doi.org/10.1145/1013208.1013209.

[41] Deepthi Amuru et al. "AI/ML algorithms and applications in VLSI design and technology". In: *Integration* 93 (2023), p. 102048. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2023.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0167926023000901.

[42] Yu Hen Hu and Sun-Yuan Kung. "Systolic Arrays". In: *Handbook of Signal Processing Systems*. Ed. by Shuvra S. Bhattacharyya et al. Cham: Springer International Publishing, 2019, pp. 939–977. ISBN: 978-3-319-91734-4. DOI: 10.1007/978-3-319-91734-4_26. URL: https://doi.org/10.1007/978-3-319-91734-4_26.

[43] AMD. *Riallto - an exploration framework for the AMD Ryzen AI NPU*. 2024. URL: https://riallto.ai/index.html.

[44] Vrije Universiteit Amsterdam. *ASTRON Nodes*. Accessed in 2025. URL: https://www.cs.vu.nl/das/ASTRON.shtml.

[45] AMD. *AMD Versal™ Adaptive SoCs*. Accessed in 2025. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal.html.

[46] AMD. *AMD Ryzen™ AI 300 Series Processors*. Accessed in 2025. URL: https://www.amd.com/es/partner/articles/ryzen-ai-300-series-processors.html.

[47] AMD. *Versal Adaptive SoC AIE-ML Architecture Manual (AM020)*. Accessed in 2025. URL: https://docs.amd.com/r/en-US/am020-versal-aie-ml/Introduction-to-Versal-Adaptive-SoC.

[48] *xdna-driver*. Accessed in 2025. URL: https://github.com/amd/xdna-driver.

[49] *Remove five-column npu1 device*. Accessed in 2025. URL: https://github.com/Xilinx/mlir-aie/issues/2371.

[50] llvm-admin team. *The LLVM Compiler Infrastructure*. Accessed in 2025. URL: https://llvm.org/.

[51] Xilinx. *MLIR-AIE*. URL: https://xilinx.github.io/mlir-aie/index.html.

[52] *Binary Formats*. Accessed in 2025. URL: https://xilinx.github.io/XRT/master/html/formats.html.

[53] Michael Larabel. *AMD's Newest Open-Source Surprise: "Peano" - An LLVM Compiler For Ryzen AI NPUs*. Accessed in 2025. 2024. URL: https://www.phoronix.com/news/AMD-Peano-LLVM-Ryzen-AI.

[54] AMD. *AI Engine Intrinsics User Guide*. Accessed in 2025. URL: https://www.xilinx.com/htmldocs/xilinx2023_2/aiengine_api/aie_api/doc/index.html.

[55] *Add worker stack size to python bindings*. Accessed in 2025. URL: https://github.com/Xilinx/mlir-aie/pull/2106/files#:~:text=Defaults%20to%201024%20bytes.

[56] *AMD Ryzen™ AI Software*. Accessed in 2025. URL: https://www.amd.com/en/developer/resources/ryzen-ai-software.html.

[57] *An end-to-end platform for machine learning*. Accessed in 2025. URL: https://www.tensorflow.org/.

[58] B. Clark. *What is quantization?* Accessed in 2025. 2024. URL: https://www.ibm.com/think/topics/quantization.

[59] Jim Holdsworth and Mark Scapicchio. *What is deep learning?* Accessed in 2025. 2024. URL: https://www.ibm.com/think/topics/deep-learning.

[60] PyTorch. *Model ensembling*. Accessed in 2025. URL: https://docs.pytorch.org/functorch/stable/notebooks/ensembling.html.

[61] A. Chia. *Open Neural Network Exchange (ONNX) Explained*. Accessed in 2025. 2024. URL: https://www.splunk.com/en_us/blog/learn/open-neural-network-exchange-onnx.html.

[62] Christiaan Boerkamp, Steven Van der Vlugt, and Zaid Al-Ars. “Tina: Acceleration of Non-NN Signal Processing Algorithms Using NN Accelerators”. In: *2024 IEEE 34th International Workshop on Machine Learning for Signal Processing (MLSP)*. 2024, pp. 1–6. DOI: 10.1109/MLSP58920.2024.10734727.

[63] Ryosuke Okuta et al. “CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations”. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[64] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[65] Kevin Babitz. *Introduction to Plotting with Matplotlib in Python*. Accessed in 2025. 2023. URL: https://www.datacamp.com/tutorial/matplotlib-tutorial-python.

[66] Claudio Brunelli, Heikki Berg, and David Guevorkian. “Approximating sine functions using variable-precision Taylor polynomials”. In: *2009 IEEE Workshop on Signal Processing Systems*. 2009, pp. 057–062. DOI: 10.1109/SIPS.2009.5336225.

[67] Perfetto. *Perfetto*. Accessed in 2025. URL: https://ui.perfetto.dev/#!/viewer.

[68] Wasser L.A. *Hierarchical Data Formats - What is HDF5?* Accessed in 2025. URL: https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5.

[69] *lofty*. Accessed in 2025. URL: https://git.astron.nl/bassa/lofty/-/tree/main/lofty.

[70] *General Matrix Multiply (GeMM)*. Accessed in 2025. URL: https://spatial-lang.org/gemm.

A

Appendix

```cpp
vector<float> image_reference2(
    const vector<float>& visR,
    const vector<float>& visI,
    const vector<float>& u,
    const vector<float>& v,
    const vector<float>& w,
    float factor,
    size_t npix_l,
    size_t npix_m,
    const vector<float>& l,
    const vector<float>& m,
    const vector<float>& n
) {
    vector<float> img(npix_l * npix_m);

    for (size_t i = 0; i < npix_l; ++i) {
        for (size_t j = 0; j < npix_m; ++j) {
            size_t index = i * npix_m + j;
            float sum = 0.0f;
            for (size_t k = 0; k < visR.size(); ++k) {
                auto A = (u[k] * l[index] + v[k] * m[index] + w[k] * n[index])*
                    factor;
                sum += visR[k] * cos(A) - visI[k] * sin(A);
            }
            float result = sum / static_cast<float>(visR.size());
            size_t index_out = (npix_l - i - 1) * npix_m + (npix_m - j - 1);
            img[index_out] = result;
        }
    }

    return img;
}
```

Listing A.1: Implementation of the All-Sky Algorithm using C++

Listing A.2: TINA layer for the All-Sky Imaging algorithm implementation in PyTorch

```python
class all_sky_image(nn.Module):
  def __init__(self,
    baselines,
    freq,
    npix_l: int,
    npix_m: int,) -> None:
```

```
    super(all_sky_image, self).__init__()

    # Generating exp matrix
    exp_matrix = tina_exp_matrix(baselines, freq, npix_l, npix_m)

    # Formatting imaginary component
    exp_matrix_imag = np.imag(exp_matrix)
    exp_matriximag_tensor = torch.from_numpy(exp_matrix_imag).float()
    exp_matriximag_tensor = exp_matriximag_tensor * -1
    shapexp = exp_matriximag_tensor.shape

    # Formatting real component
    exp_matrix_real = np.real(exp_matrix)
    exp_matrixreal_tensor = torch.from_numpy(exp_matrix_real).float()
    shapexp = exp_matrixreal_tensor.shape
    exp_matrix_comp = torch.concat((exp_matrixreal_tensor, exp_matriximag_tensor),
        dim=1) # concatenated exp matrix, with shape [2, pixels, 96, 96]

    self.matrix_mult = inner_conv(exp_matrix_comp)


 def forward(self, x):
  # print("shape of input Sky_imager_TINA_real: ", x.shape)
  outputcomp = self.matrix_mult(x)
  return outputcomp
```

Listing A.3: TINA layer convolution layer inside the TINA layer implementation of the All-Sky Imaging algorithm.

```
class inner_conv(nn.Module):
 def __init__(self, matrix) -> None:
    super(inner_conv, self).__init__()
    shape = matrix.shape

    self.batch, self.channels, self.height, self.width = shape

    self.conv_layer = nn.Conv2d(2, self.batch, bias=False, kernel_size= (self.
        width,self.height), stride= (1, 1), groups= 1)
    weightsconv = matrix

    self.conv_layer.weight.data = weightsconv

 def forward(self, x):
  shape = x.shape
  out = self.conv_layer(x)
  out = out.view(1, 1, self.batch, 1)

  return out
```

Listing A.4: Numpy implementation of the exponential matrix used for the TINA implementation of the All-Sky Imaging algorithm

```
def tina_exp_matrix(
   baselines,
   freq,
   npix_l: int,
   npix_m: int,
):
   exp_matrix = np.zeros((npix_l*npix_m,1 , baselines.shape[0], baselines.shape
       [1]), dtype=np.complex64)
   img = np.zeros((npix_l, npix_m), dtype=np.complex128)
   l, m = np.meshgrid(np.linspace(-1, 1, npix_l), np.linspace(1, -1, npix_m))
   n = np.sqrt(1 - l**2 - m**2) - 1
```

```
    for l_ix in range(npix_l):
        for m_ix in range(npix_m):
            exp_matrix[l_ix * npix_m + m_ix, :, :, :] = np.exp(
                    -2j
                    * np.pi
                    * freq
                    * (
                        baselines[:, :, 0] * l[l_ix, m_ix]
                        + baselines[:, :, 1] * m[l_ix, m_ix]
                        + baselines[:, :, 2] * n[l_ix, m_ix]
                    )
                    / SPEED_OF_LIGHT
            )
    return exp_matrix
```