Signal Processing with AMD Adaptive Compute Acceleration Platform (ACAP) for Applications in Radio Astronomy

VICTOR VAN WIJHE, University of Twente, Netherlands

High-performance computing (HPC) has become essential due to the need to process enormous amounts of data quickly and accurately amongst others in the field of radio-astronomy. The Versal Adaptive Compute Acceleration Platform (ACAP) is a recent and promising alternative addition to the field of domain-specific accelerators. This thesis conducts a design space exploration of the Versal ACAP AI Engines using a polyphase filter. This study aims to assess the efficiency of utilizing the Artificial Intelligence engines within the ACAP for signal processing tasks in radio astronomy. We achieved this by deploying multiple streaming-based polyphase filter designs on the ACAP and conducting an evaluation of the AMD Tooling and AI Engines' performance. A design space exploration was conducted to discover the existing libraries applicable to polyphase filters. The available FIR library from AMD reached a throughput of 208 M samples/s for a single branch but could not be effectively scaled to the requirements of the use case. The AMD FFT library managed an impressive throughput of 220 M samples/s on a single AI Engine and is used for the polyphase application. To overcome the AMD FIR library limitation, five distinct polyphase filter kernels have been designed ranging in effectiveness. A throughput was reached of 312 M samples/s when the filter is deployed on four AI Engines, however greater throughput could be reached by utilizing more AI Engines. Finally, a polyphase filter library is created adaptable to a wide range of use cases. Despite promising advancements such as the release of AIE-ML with enhanced performance and a 16-bit bfloat, the platform's maturity of tooling remains a significant hurdle, as available features can not be optimally utilized. This work provides a foundation for future research and development when using the AMD Versal ACAP.

Additional Key Words and Phrases: Polyphase filter, DSP, ACAP, AI Engine, Radio astronomy

1 INTRODUCTION

In today's fast-paced world of computing, integrating specialized hardware has become crucial for boosting performance and efficiency in various tasks. High-performance computing (HPC) has become essential in fields like genomic sequencing, autonomous vehicles, and virtual reality due to the need to process enormous amounts of data quickly and accurately. Another high-performance application could be found in radio telescopes. Before stating the application used with radio telescopes, some background information is needed. There are two main kinds of radio telescopes: dish-style telescopes consisting of a single large dish that reflects the incoming radio waves to a focal point where the data is collected, and array-style telescopes that are made up of several to many smaller dishes or antennas spread out over a region that are computationally combined to effectively create a telescope with a larger collecting area and diameter [1]. This work focuses mostly on an array-stile telescope. A station, consisting of multiple individual antennas, generates a vast amount of data which is sent over public and private Ethernet to a central processor. To ease the load on the network and reduce the cost, edge devices are used to pre-process the collected data close to the antennas. The data is filtered to create finer-grained frequency channels and is then beamformed to the section of the sky that researchers are interested in, reducing the data rate. The first stage of the pre-processing pipeline uses a polyphase filter (PPF) to divide the sampled data in finer grained frequency channels. Polyphase filters have applications in astronomy [2] [3], wireless communication [4] [5] [6], radar signal processing [7] and many other signal processing domains.

The Netherlands Institute for Radio Astronomy (ASTRON) operates among others the Low Frequency Array (LOFAR)[2] radio telescope. It utilizes many stations, spread over Europe, with each station Besides the LOFAR radio telescope the requirements of two other systems from the same domain with similar characteristics are taken into

Author's address: Victor van Wijhe, v.p.vanwijhe@student.utwente.nl, University of Twente, P.O. Box 1212, Enschede, Netherlands, 43017-6221.

account to assess scalability and adaptability of the platform. The Spectrometer [8] [9], and the Microwave Kinetic Inductance Detector I/II[10] are introduced in more detail in section 2.

As the polyphase filter is deployed on edge devices compute density, efficiency and power consumption is very important. This drives the need to do research on different architectures to determine its effectiveness. Polyphase filters have already been implemented on many different architectures, including FPGAs [11] [12] [13], GPUs [14] [15], multi-core processors [14] [15] and special purpose Digital Signal Processors, such as the SHARC [16]. The Versal Adaptive Compute Acceleration Platform (ACAP) is a recent and promising alternative addition to the field of domain-specific accelerators. It combines traditional FPGA programmable logic with adaptable hardware accelerators, a network-on-chip, and an array of processing engines to create a holistic platform for a wide range of applications. The most novel features of the ACAP are the Artificial Intelligence engines, a grid of processing engines with vector units capable of running individual kernels and communicating with each other. Implementing the polyphase filter pipeline will give more insights into the performance, capabilities, and ease of development of the AMD Versal ACAP. Currently the Versal ACAP is mostly utilized in the Machine Learning paradigm such as with Graph Convolutional Network (GCN) accelerator that utilizes both the programmable logic and AI engines [17], Convolutional Neural Network (CNN) [18] [19] [20] and computer vision [21]. An important part of the convolutional networks are matrix multiplications, as such research has been done with regards to matrix multiplications specific to the Versal ACAP [22] [23] [24].

In this work we aim to evaluate how effectively the AMD Versal Adaptive Compute Acceleration Platform's Artificial Intelligence engines can be utilized for signal processing applications in radio astronomy. To do so, the emphasis will be on implementing and evaluating streaming-based polyphase filter designs for the LOFAR radio telescope. The contribution of this work is two-fold:

- We present multiple streaming-based polyphase filter designs implemented on the AMD Versal ACAP using the artificial Intelligence engines. By making the polyphase filter design compatible with the requirements of the LOFAR the opportunity is created to prepare and channelize the data for the beamformer.
- An evaluation is provided of the AMD Tooling and AI Engines. This evaluation will consist of available libraries, functionality and possible future improvements.

The remainder of this work is organized as follows: section 2 describes the basic principles of a radio telescope, polyphase filter and the architecture of the ACAP. section 3 presents related work on previous implementations on the ACAP and of the polyphase filter. section 5 describes the different implementations and results of the polyphase filter on the ACAP. We conclude this work in section 6.

2 BACKGROUND

A radio telescope operates by receiving radio waves collected by an antenna that converts the waves into an electrical signal. This analog electrical signal is then digitized using Anolog to Digital Converter (ADC) allowing it to be further processed into data products that are used by astronomers. There are three main types of radio telescopes: parabolic dish telescopes and phased array telescopes, or a combination.

A dish telescope consists of a single large parabolic dish that reflects all the incoming radio waves to a focus point where usually a so-called feed, the antenna, is placed that collects the data. The sensitivity of a dish telescope is limited by the size of the dish. As they are only able to reflect so much, this means that the telescopes are hard to extend and improve during their life. The dish also has to be pointed in the direction of the sky for a specific observation. This results in complicated elevation and azimuth control to move the heavy dish slowly in the correct direction.

An alternative would be the phased array telescope, this style of telescope consists of many smaller-scale antennas that with the help of beamforming and interferometry can perceive a section of the sky. The phased array telescope combines multiple antennas into a single virtual telescope allowing it to increase the surface and diameter, focusing only on a specific area of interest by post-processing the collected data. Phased array telescopes are also relatively low cost allowing them to be extended over their lifetime, resulting in great adaptability. Figure 1 shows an overview of a generalized phased array radio telescope system.

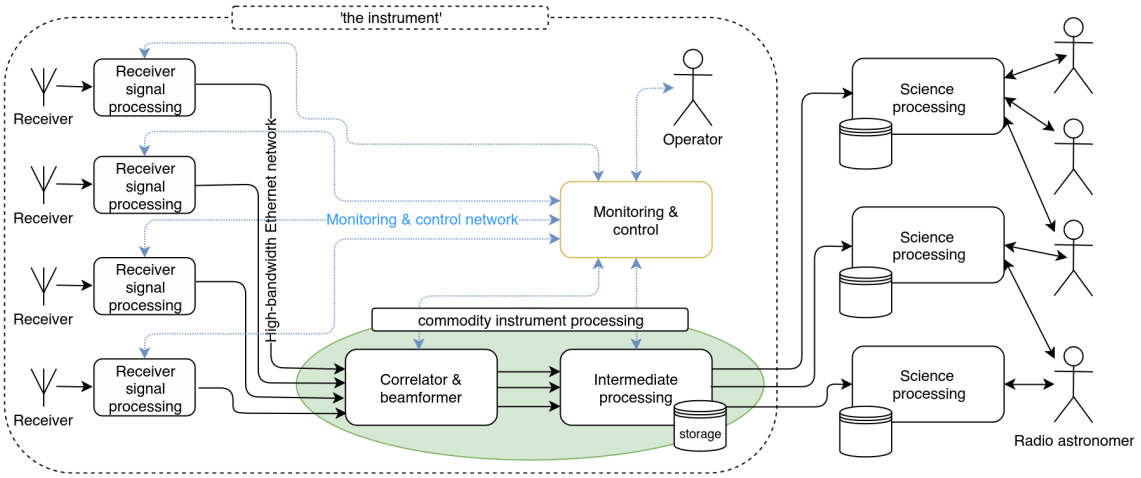


Fig. 1. Top level overview of a generic phased array radio telescope [25].

In recent years a combination of phased array and parabolic dish telescopes have been introduced such as the WSRT[26], ALMA [27] and the SKA (mid) [28] combining the sensitivity of the dish telescopes with the resolution and flexibility of the phased array telescopes. Three radio telescopes will be used to extract requirements for the polyphase filter: LOFAR [2], the main focus of this work and phased array telescope, the Spectrometer [8] [9], and the MKID I/II[10]. Only the basic requirements for the spectrometer and MKID I/II are covered in this research here since they are not the main focus of our work. They are included to show how well our research can be applied to different requirements.

Each LOFAR station consists of two different groups of antennas: 96 Low-Band Antenna (LBA), optimized for 15-80 MHz, and either 48 or 96 High-Band Antenna (HBA), optimized for 110-240 MHz. Figure 2 shows the antennas used for

LOFAR LBA and HBA. The antennas are dual polarization antennas, so each antenna yields 2 signal inputs. The signal from each polarization is sampled using a 14-bit ADC with 200 Mega Samples Per Second (MSPS), resulting in an input data rate of 14b * 200M = 2.8 Gbit/s per signal input. Hence, a LOFAR station with 96 dual polarization LBA generates a total input data rate of 2.8 Gbit/s * 2 * 96 = 537.6 Gbit/s.

Fig. 2. the LOFAR HBA *(left)* and LOFAR LBA *(right)* in Tautenberg. Copyright - ASTRON

The Spectrometer is a component of a heterodyne receiver, a signal-processing front end for radio telescopes in the mm and sub-mm regime (100 GHz to 4 THz). It is developed for use with APEX, GREAT [3] and CHAI [29]. Compared to LOFAR it has a much higher sampling frequency and fewer antennas. The input data is sampled with 8 GSPS, using an 8-bit ADC, resulting in an input data rate of 64 Gbit/s.

The A-MKID is a sub-millimeter super-conduction photon camera feed that is developed for deployment on the APEX telescope. Both versions of the A-MKID are sampled with 4.1 GSPS using a 12-bit ADC, resulting in an input data rate of 98.4 Gbit/s. Table 1 shows an overview of the input requirements of the three systems.

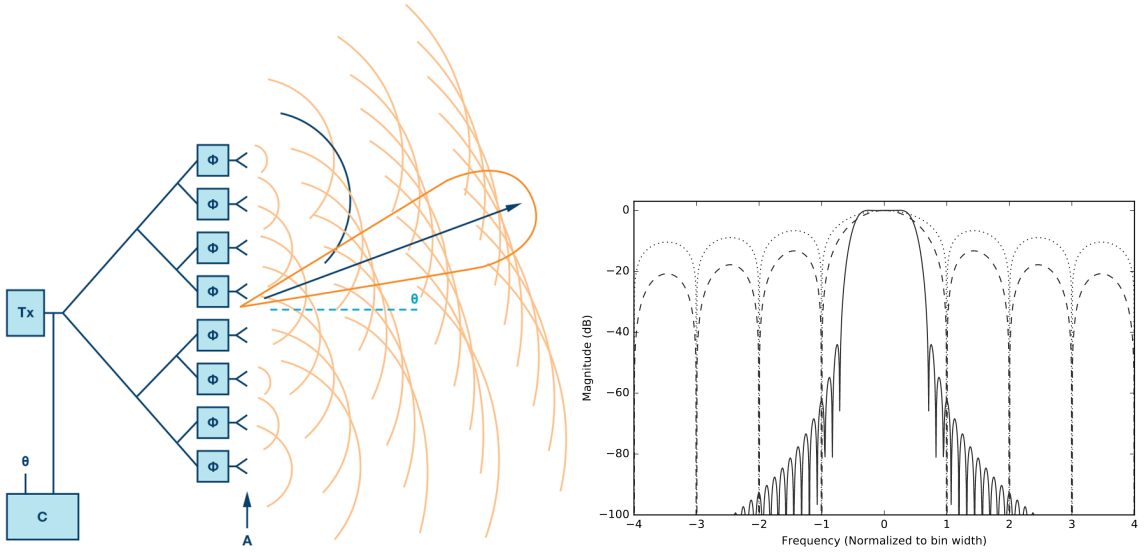
Table 1. Input requirements of the LOFAR, Spectrometer, A-MKID I/II.

	LOFAR 2.0	Spectrometer	A-MKID I/II
Sample rate	200 MSPS	8 GSPS	4.1 GSPS
Input bit width	14 bit	8 bit	12 bit
Receivers per device	96 * 2	1	2
Input data rate	537.6 Gbit/s	64 Gbit/s	196.8 Gbit/s

2.1 Polyphase filter

In the three use cases described above, the data arriving at the antenna of the radio telescopes is sampled by an ADC and pre-processed on an FPGA. The sampled data is first filtered and beamformed. The first stage filter creates narrower frequency channels to allow more efficient beamforming and processing further on. This first-stage filter is often implemented with a polyphase filter. The Beamformer introduces small delays between the data from different antennas while combining them to effectively focus on a part of the sky and reducing the amount of data significantly. Figure 3a

shows this principle in reverse. In this study, we focus on this first stage filter as it forms a representative challenge for applications of the ACAP AI Engines (AIE) in the context of radio astronomy.



(a) Beamforming using delays between antennas. Source: [30].

(b) Comparison of the channel response of an ACS (dotted line), FTF (dashed line) and an 8-tap, Hann-windowed PFB (solid line). Source: [31].

Fig. 3

A polyphase filter consists of an FFT and Finite Impulse Response (FIR). The FFT transform converts the time domain digitized input signal to frequency channels. However, the FFT introduces spectral leakage in the resulting frequency channels, due to the assumption that the signal is periodic. By applying a FIR to each output channel of the FFT, the spectral leakage is reduced, as shown in Figure 3b. It also introduces a decimation factor by filtering out data outside of the specified band. The second noble identity states that the FIR bandpass filter and Fourier transform can be interchanged. Implementing the FIR filters before the FFT increases efficiency, in the implementation, as the FFT processes less data. Incoming digitized samples are round-robin distributed over the multiple FIR filter to load balance the system. This combination of an FIR and FFT is referred to as a polyphase filter, shown in Figure 4. A polyphase filter can be specified using three parameters: the number of FIR branches, the depth of the FIR, and the number of spectral channels. FIR branches are the number of parallel FIR filters that feed into the FFT. A FIR filter is, in principle, a series of history samples that are multiplied by coefficients (weights) and summed. The FIR depth determines the number of history samples within the FIR filter. The number of required spectral channels determines the number of frequency bins that constitute the frequency domain representation, for complex valued input samples, the number of spectral channels is equal to the number of branches. Whereas for real valued input samples, the number of spectral channels produced by the FFT is half the number of branches.

All three of the given use cases (LOFAR, Spectrometer, A-MKID) provide different requirements for their spectral channels, FIR branches, and FIR depth. The input data can be real or complex, allowing for a specialized FFT, and

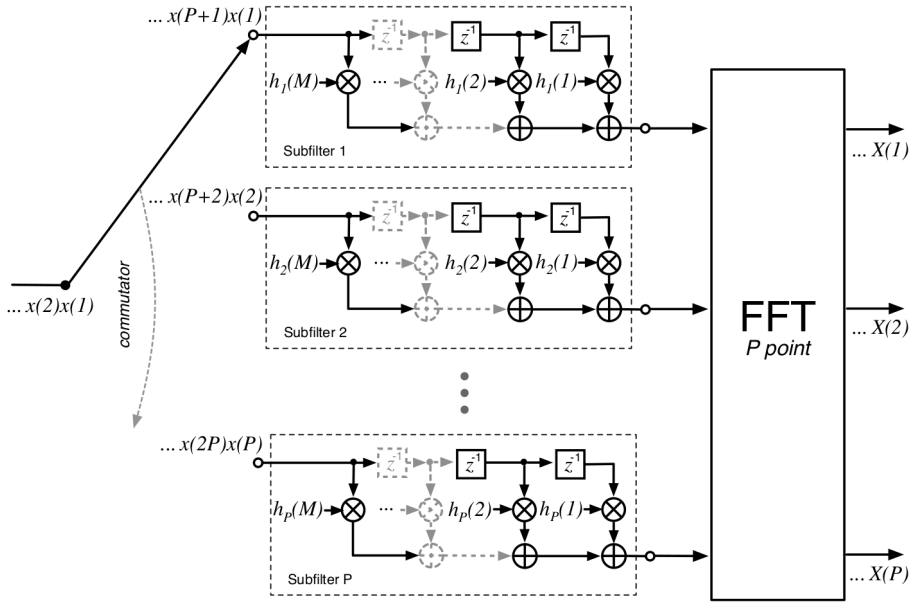


Fig. 4. An overview of the Receiver signal processing block. Data is round-robin distributed over multiple FIR filters and the output of all filters is the input of an FFT. source: [31].

reducing the number of spectral channels, in the case of real data. The requirements for each of the use cases is given in Table 2.

Table 2. Polyphase filter requirements for the LOFAR/Spectrometer/MKID.

	LOFAR 2.0	Spectrometer	A-MKID I	A-MKID II
Input	Real	Real	Complex	Complex
FIR branches	1024	64 k	1024 k	2 k
FIR taps	16	4	0	4
Spectral channels	512	32 k	1024 k	2 k

2.2 ACAP

Now that we've outlined the necessary conditions for running the use cases, let's take a closer look at the system architecture that will host and manage these operations. In the current landscape of signal processing, the prevalent use of FPGAs has been driven by their exceptional interfacing capabilities with ADCs. While FPGAs excel in this role, our exploration into the efficiency of preprocessing tasks has led us to consider the untapped potential of ACAPs. ACAPs represent a novel paradigm where the interfacing prowess of FPGAs converges with the flexibility of AI Engines, featuring vector units designed for signal processing. This section discovers the possibilities of ACAPs.

Within the Versal AI Core series, the fusion of three distinct architectures: Programmable Logic (PL), Programmable Subsystems (PS), and AI Engines (AIE) form the cornerstone of its computational capabilities. The integration of these elements presents a comprehensive approach to adaptive computing, allowing for flexibility and performance. To

facilitate interoperability among these components, AMD has developed a Network on Chip (NOC). This networking infrastructure serves as the connective tissue, enabling efficient data exchange between the Programmable Logic, Programmable Subsystems, and AIE within the ACAP. The three distinct architectures combined with the NoC are shown in Figure 5.

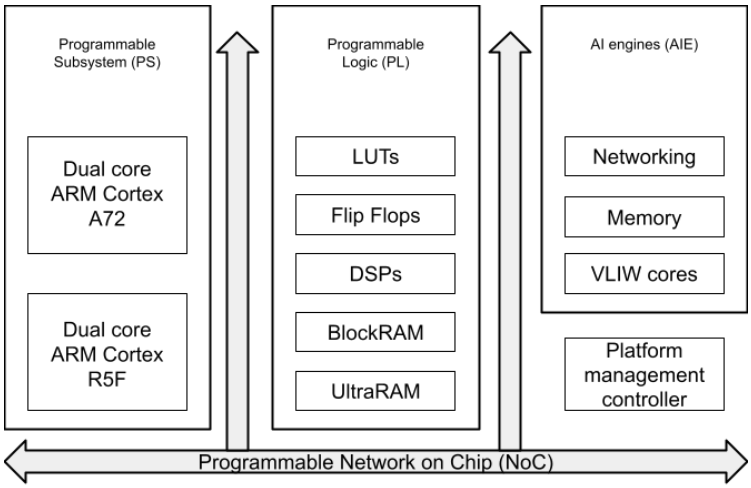


Fig. 5. Versal ACAP architecture. Adapted from: [32].

From the three architectures, this study focuses on the capabilities of the AI Engines (AIE). The technical architecture of the AIE on the ACAP is defined by a systolic array, where each tile is linked through an associated interface tile and a routing network connecting it to neighboring tiles. The collective assembly of all tiles forms an 8x50 AIE array, totaling 400 tiles. Figure 6 shows a part of the systolic array indicating individual AI Engines, the cascade interface indicated by blue arrows (single direction arrows) and DMI/stream interface by the red arrows (dual direction arrows), both expanded upon below.

Each AIE tile contains dedicated memory and a Very Long Instruction Words (VLIW) processor, paradoxically referred to as the AIE. To ensure clarity and coherence in subsequent chapters of this work, it is imperative to establish a clear terminology: the collective assembly of all tiles forms the AIE array, while an individual tile shall be designated as an AI engine. In addition to its computational capabilities, each AIE tile boasts a robust memory architecture, containing eight memory banks, each with a capacity of 4096 bytes. This configuration results in a total memory capacity of 32 KiB per AIE tile. To facilitate data access, each AIE tile supports multiple interfacing options, also shown in Figure 7:

The Data Memory Interface (DMI) is designed for processing large blocks of data, providing a mechanism for handling datasets reaching 80 GB/s read and 40 GB/s write.

$$((256 * 2)/8) * 1.25GHz = 80GB/s \;\; read$$

$$(256/8) * 1.25GHz = 40GB/s \;\; write$$

Meanwhile, the stream interface is optimized for smaller chunks of data arriving at regular intervals, making it ideal for real-time applications with a 5 GB/s read and write. From now on when mentioning streams we refer to the stream

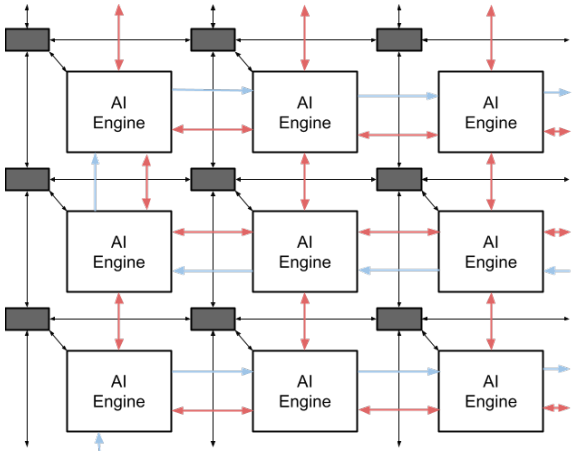


Fig. 6. Representation of the AI Engines systolic array. Adapted from: [33].

interface.

$$(32/8) * 1.25GHz = 5GB/s \quad read\ or\ write$$

Propagation of (partial) results is enabled by the Cascade interface using an accumulator datatypes with a throughput of 60 GB/s. Regular datatypes are also able to utilize the cascade interface after being converted to accumulators.

$$(384/8) * 1.25GHz = 60GB/s \quad read\ or\ write$$

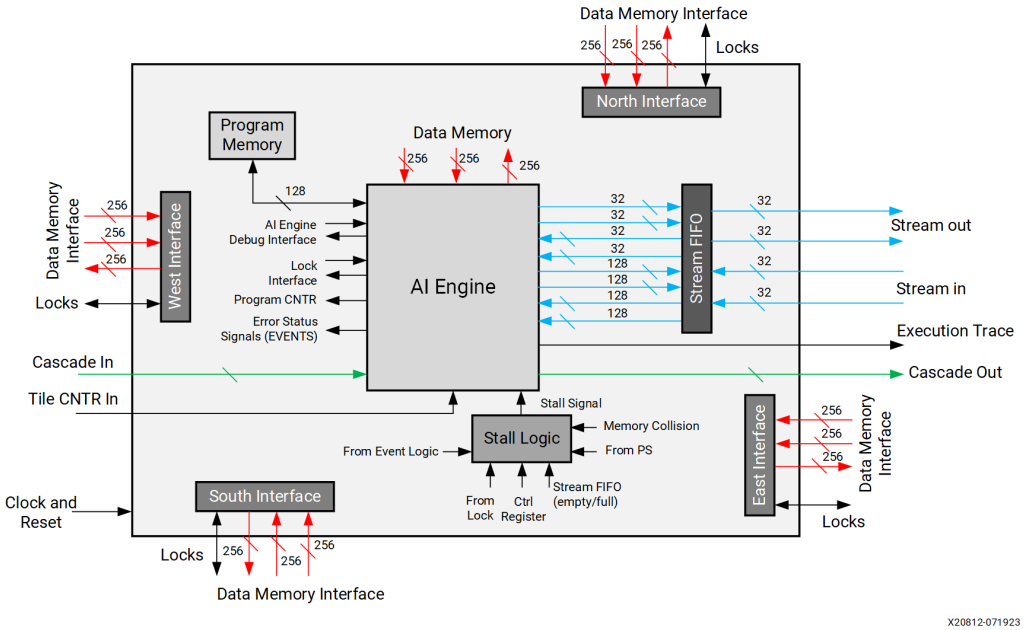


Fig. 7. The available interfaces on an AI tile. Source: [34].

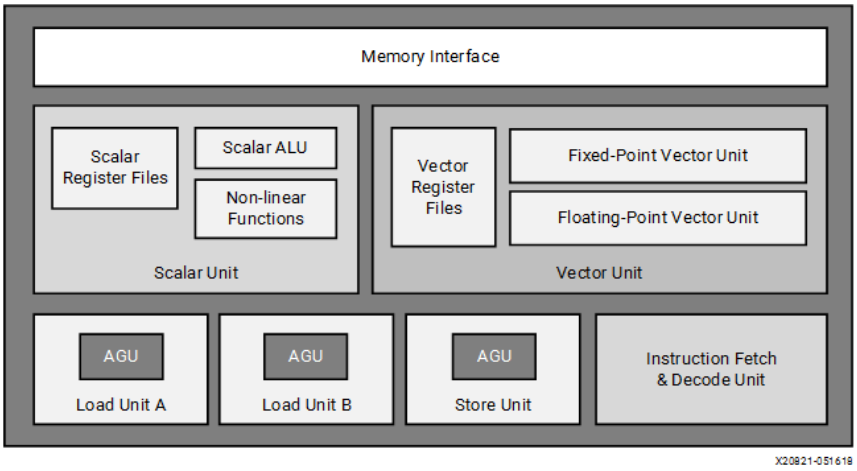


Fig. 8. Functional overview of the AI engine architecture. Source: [35]

This flexible interfacing approach equips the AIE tile with the adaptability needed to cater to diverse data access patterns and optimize overall system performance. In addition to the memory embedded within the AIE tile itself, the processor has extended memory access capabilities by tapping into the memory of, at most, three neighboring tiles. This design choice expands the theoretical memory limit for the processor to 128 KiB. Distinguishing the ACAP are various configurations known as speed grades, each specifying a clock speed bound to the operating voltage. These speed grades provide flexibility to tailor the ACAPs performance to specific application requirements. In the context of this paper, the selected ACAP device is the VC1902 on the VCK5000 accelerator card, characterized by a clock speed of 1.25 GHz.

Each AIE tile encompasses its dedicated processor operating at a clock speed of 1.25 GHz. This processor is a 128-bit VLIW architecture with Single Instruction Multiple Data (SIMD) capabilities. The processor combines a 512-bit vector unit and a 32-bit scalar unit, shown in Figure 8, both of which can be utilized in a single VLIW instruction. Notably, the processor executes varying numbers of operations per clock cycle, depending on the datatype in use. For instance, when processing a 32-bit floating-point data type, the vector processor achieves 16 operations per clock cycle. Consequently, the processing power available on a single AIE is 20 Giga Operations Per Seconds (GOPS) when using a floating-point.

When clarifying the architectural details, it's essential to note that the scalar processor within each AIE tile lacks a dedicated floating-point unit. The vector processor is equipped with registers specifically designated for vectors and accumulators. These hardware registers are inherently 128-bit wide, with the flexibility to combine two 128-bit registers to form a 256-bit register. Further amalgamation allows the creation of 512-bit registers, and these, in turn, can be grouped to generate expansive 1024-bit registers. This modular structure gives the vector processor scalable and adaptable register configurations. However, owing to the fixed size of the registers, each datatype exhibits a distinct range of allowable sizes represented in Table 3, underscoring the nuanced capabilities of the ACAP's vector processor in handling various data types.

The AI engines within the ACAP AIE array seamlessly integrate two programming paradigms: object-oriented programming, where algorithms are structured to process data using functions known as kernels, and dataflow programming, which models a program as a directed graph. In the context of the AI engine array, this involves

Table 3. The allowed sizes of vector and operations per clock-cycle for each datatype

Data types	Allowed sizes of vector	Operations per clock-cycle
int8	16/32/64/128	248
int16	8/16/32/64	128
int32	4/8/16/32	16
uint8	16/32/64/128	248
float	4/8/16/32	16
cint16	4/8/16/32	16
cint32	2/4/8/16	2
cfloat	2/4/8/16	2

establishing interface connections, initializing kernels, and specifying the allocation of kernels to individual AIE tiles. This dual paradigm approach enhances flexibility and efficiency in programming, allowing for the streamlined execution of complex signal processing applications on the ACAP architecture.

3 RELATED WORK

In the exploration of related work, this chapter first delves into the state of the art, specifically examining implementations of polyphase filters on FPGAs or GPUs. The focus here is to illuminate existing approaches allowing us to compare our performance against these implementations. Following this investigation, the discourse shifts towards signal processing applications tailored for the ACAP AIE array. This section aims to provide insights into the available documentation on the performance, design space exploration, and user experience of the ACAP. Finally, the related work extends to an overview of existing software libraries available for AI engines. By examining the available tools and libraries, the chapter seeks to identify the support structures in place for developers working with the ACAP architecture. Together, these three strands of related work aim to establish a contextual foundation for the subsequent chapters.

First looking at existing PPF implementations on FPGA, J.P. Smith et al.[36] stands out because they designed and implemented a high-performance polyphase channelizer on a Xilinx (now AMD) RFSoC (ZCU111). Their solution, boasts a 4 GSPS, 4096-branch, 8-tap, 2/1 oversampled polyphase filter. Utilizing Xilinx FIRs and FFTs, The bandwidth and polyphase filter specifications are comparable to the requirements of this project. In the end, this approach was resource-constrained by the FIR filter LUTRAM. Additionally, L. H. Arnaldi[37] made a significant contribution by implementing a polyphase filter channelizer on a ZYNQ FPGA, achieving a resource utilization of 50% with a 125 MHz input sample rate. These studies exemplify effective implementations of polyphase structures on FPGA architectures.

Within the context of GPU-based signal processing, a significant achievement is attributed to [38], who developed a polyphase filter for the MeerKAT radio telescope, using a GeForce RTX 300 Ti. The system can process four 2 Gsample/s antennas or a single high-bandwidth 6.2 Gsample/s antenna. The spectrometer antennas generate 8 Gsamples/s making it incompatible with our main use case. Another difference is the placement of the polyphase filter as it is located in the back end instead of the front end as with the LOFAR telescope. N. Ragoomundun et al.[39], developed a polyphase filter featuring 8 taps and 16384 channels using an Nvidia GeForce GTX Titan X (Maxwell). This GPU-powered solution demonstrated remarkable prowess by efficiently processing data at a rate of 12 GiB/s, with the performance constrained by the PCIe x16 interface.

Currently, very few applications have been published using the Versal ACAP due to the relatively recent introduction and scarce product availability. A. Al-Zoubi et al. [18] implemented a CNN implementation on a Versal ACAP and showed that it could outperform a high-end CPU, GPU, and the Zynq UltraScale+ DPU. However, the peak performance of the ACAP can only be achieved once the CNN is fully supported by the DPU on the ACAP. The study used Vitis AI and Tensorflow to train and deploy the CNN, as such not providing much insight in using the AIE array architecture in general. N. Perryman et al. [40] makes a comparison between the processing elements within the VCK190 by running multiple different applications on each element such as the programmable logic, programmable subsystems, and AI engines. The MTSE application is closest to our application, as it is a DSP application containing many FFTs. The AI engines showed performance benefits, especially for the MTSE algorithm. N. Perryman et al. do not mention the development environment on which the CNNs and the MTSE algorithm are developed, but we can assume that they used Vitis as the runtime ratio is mentioned. Besides the runtime ratio, no other remarks were made about design space exploration or user experience, only mentioning that they chose a runtime ratio of 0.6 based on reference designs. Also,

the implementations used are not freely available, thus making it hard to reproduce the work for a different domain. J. Zhuang et a. [22] focuses on a method called CHARM to increase efficiency when the VLIW of the AI engines are underutilized when smaller datatypes than natively supported are used. By implementing CHARM a considerable increase in performance was achieved for CNNs, reaching an impressive 94.70% resource utilization on a single AI engine using the AI intrinsics. They do introduce a more detailed discussion of the AI engines programmability by explaining packet switching, however many interfaces are not discussed. It is also not directly applicable as they focus on matrix multiplications specifically, where we focus on a use case of applied vector processing. The limited number of public applications developed on the ACAP platform shows that there is still a lot to gain in this field.

A signal processing application implementation hosted on an ACAP, that is comparable to applications in radio astronomy, is a beamformer implementation by Frederik Johansson & Lukas Ögnelod's master thesis project[41]. Beamforming is the second stage of the LOFAR pre-processing pipeline. The work by Frederik Johansson & Lukas Ögnelod leverages AI engines to achieve a throughput of 7.8 Mega samples per second. This work was very informative on the capabilities of the ACAP, however, the throughput of the system is not in the same magnitude as for our system so we might run into different challenges.

In collaboration with our work, a bachelor thesis project was carried out at The Otto von Guericke University Magdeburg by Vincent Sprave. The work focuses on the data transport from the ACAP Programmable Logic to and from the AIE array, whereas our work focuses on the implementation of an application on the AIE array.

AMD provides an extensive list of libraries and kernels that can be integrated into projects such as libraries for linear algebra, vision, DSP, and many more[42]. This work uses the DSP library as it provides an FIR filter [43] and FFT [44] implementation that could be used as a base to develop the specific FIR and FFT required for the polyphase filter for the LOFAR use case. The AI Engine Kernel and Graph Programming Guide [45] and AI Engine Programming Environment User Guide [46] gives an extensive explanation of how to program and deploy for AI Engines. AMD gives two methods to utilize the power of the vector processor: AI Engine intrinsics [47] and the AI Engine API[48], both provide descriptions of datatypes and functions, but the references are not interchangeable.

Currently, there is no assessment of the available libraries, also reports on the user experience with the AI engine platform are limited. This work will present a thorough design space exploration of the architecture with respect to compute, storage, and interfacing taking a polyphase filter for the LOFAR telescope as a use case.

4 THEORETICAL ANALYSIS

The theoretical analysis of this study begins with the available compute resources on both a single AIE tile and the entire AIE array. By quantifying the processing power of these components, the research aims to establish a baseline for evaluating the feasibility of executing FIR and FFT algorithms. Subsequently, the compute requirements of ideal algorithms for FIR and FFT will be calculated, allowing for comparative analysis against the available compute on the AIE array. This comparison will illuminate the potential for optimal algorithm execution within the constraints of the ACAP architecture.

Building upon the background information, the computational capabilities of the AI engine on a single AIE and the entire array of 400 AIE tiles have been established. Specifically, for a 32-bit floating-point datatype, the AI engine can execute 16 operations per clock cycle, equating to 8 Multiply-Accumulate (MAC) operations. With a clock speed of 1.25 GHz, this results in a substantial compute power of $2x10^{10}$ operations per second (OPs/s) for an individual AIE. When scaled across the entire array, the cumulative compute reaches an impressive $8x10^{12}$ OPs/s.

$$1.25 \text{ Ghz} * 400 \text{ AIE} * 16 \text{ operations} = 8 * 10^{12} \text{OPS/s} \tag{1}$$

Moving forward, these computational metrics are integrated into roofline plots [49]. The plot consists of two main components: the performance ceiling or "roofline" which represents the maximum achievable performance of a system given its computational capabilities, and the performance profile of the algorithm or program being analyzed. The x-axis represents operational intensity, which is a measure of computation per unit of memory movement. The y-axis represents performance in Giga Operations Per Seconds, where the horizontal lines indicate the maximum available performance on the device. The plot will be populated with points that indicate whether an implementation is within the available compute and bandwidth of the device. This visual representation will serve as a tool for assessing the implementation of algorithms, indicating whether they are compute-bound (when a point is above the horizontal roofline) or interface-bound (when a point is above the diagonal roofline, but below the horizontal roofline) and how close it operates to the theoretical peak performance of the system. The roofline plot provides an overview of the ACAP's performance envelope.

Figure 9 displays two distinct groupings: one for a single AI engine (the lower grouping) and one for the entire AI array (the upper grouping). Each grouping comprises three different data types: int8, int16, and floating point, highlighting the differences in computational capacity based on the data type. For our specific use cases, we're only concerned with the floating point, so the int8 and int16 rooflines will be left out in future plots. Apart from the data types and groupings, the image also indicates the input interface used by the single AI Engine. As explained in section 2, an AI Engine employs two primary methods of data transfer: the Direct Memory Interface (DMI) with a throughput of 80 GiB/s and the Stream interface with a throughput of 5 GiB/s. The variation in throughput could potentially impact whether the data processing is interface-bound.

The minimal computational requirement for an asynchronous FIR filter is determined by the depth of the filter, with each history sample necessitating two operations, multiplication by its corresponding coefficient and summation to the total. The LOFAR application is characterized by a depth of 16. The achievable performance is calculated by multiplying this operation count with the input sample rate of the pipeline, which is 200 million samples per second. Consequently, the attainable performance for the asynchronous FIR filter in the LOFAR application stands at $6.2 * 10^9$ Operations Per Second (OPS).

$$200 \text{ M samples/s} * 31 \text{ operations/sample} = 6.2 * 10^9 \text{ OPS} \tag{2}$$

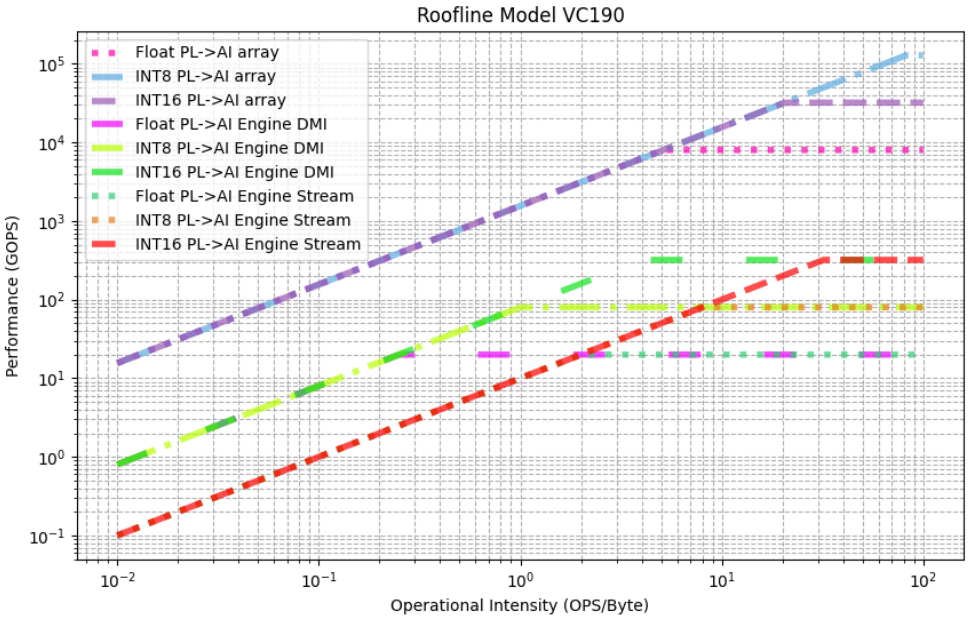


Fig. 9. Roofline plot of the int8/int16/float datatypes on the VC1902.

This computation sets a foundational benchmark for evaluating the feasibility and efficiency of implementing the FIR filter on the ACAP AIE array.

Calculating the minimal computational requirement for the Fast Fourier Transform (FFT) involves a more intricate formula, as provided by [50]: $5Nlog_2(N)$ to compute the total number of operations needed. By substituting 1024 branches for N, this results in 51200 operations per calculated FFT. Since a single FFT is calculated for every 1024 samples, the overall operations per second can be determined, resulting in $1.0 * 10^{10}$ operations per second.

$$200 \text{ M samples/s} * \frac{51200}{1024} \text{ operations} = 1.0 * 10^{10} \text{ OPS} \tag{3}$$

Table 4 presents the outcomes derived from applying the specifications of the MKID and Spectrometer with the same formulas. Figure 10 shows the placement of these ideal applications on the roofline plot. Both the ideal FIR filterbank and FFT of the LOFAR fall within the compute capabilities of a single AIE. The FIR executes 7.75 operations per byte, determined by the depth of the FIR, and is therefore not compute bound.

By aggregating the operations per second of both the ideal FIR and ideal FFT and considering operational intensity, the characteristics of the ideal polyphase filter have been determined and are depicted in the roofline plot. The roofline plots demonstrate that the computational demands required to meet the specified requirements fall within the capacities of the ACAP AIE array. The points indicate that an ideal polyphase filter with the requirements of the LOFAR system could be deployed on a single AIE tile. The combined 192 receivers could theoretically be deployed on the entire array.

Table 4. Resulting computational effort

	LOFAR 2.0	Spectrometer	A-MKID 1	A-MKID 2
single FIR	31	7	–	7
combined FIR	$3.1 * 10^4$	$4.5 * 10^5$	–	$1.4 * 10^4$
FFT	$5.1 * 10^4$	$5.1 * 10^6$	$1.0 * 10^8$	$1.1 * 10^5$
polyphase filter	$8.3 * 10^4$	$5.6 * 10^6$	$1.0 * 10^8$	$1.2 * 10^5$
polyphase filter/s	$1.6 * 10^{10}$	$7.2 * 10^{10}$	$2.0 * 10^{10}$	$4.0 * 10^{10}$
polyphase filter/s x receiver/device	$3.1 * 10^{12}$	$7.2 * 10^{10}$	$4.0 * 10^{10}$	$7.9 * 10^{10}$

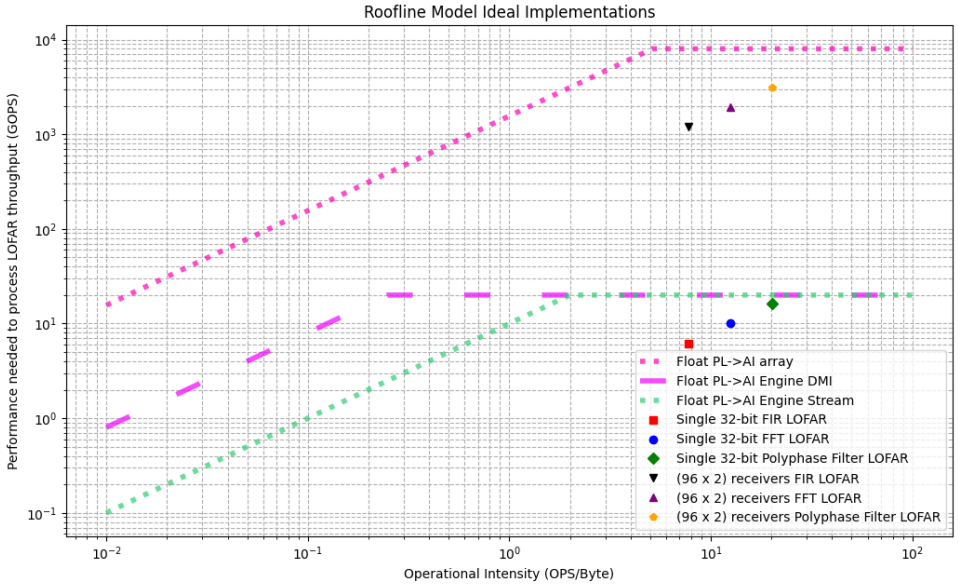


Fig. 10. The Placement of the ideal polyphase filter implementation on the roofline plot. The LOFAR FIR + FFT should just fit on a single AI engine

5 IMPLEMENTATION

Following the determination of the ideal implementations, the research transitions to design space exploration. Initially, an exploration of libraries provided by AMD will be conducted to identify existing tools and resources that align with the computational demands of the FIR and FFT algorithms. In cases where the available libraries fall short of the project's specific requirements, custom software will be developed to meet these demands. This adaptive approach ensures a thorough exploration of the design space, leveraging both existing resources and custom solutions to maximize the potential of the ACAP AIE array in signal processing applications. When suitable kernels have been identified they will be integrated into a dataflow application combining the different kernels in a functional application. Finally, a generic polyphase filter library is created applicable to a wide range of requirements.

In our implementation, the toolchain comprises Vivado for the creation of a base platform, facilitating the integration of programmable logic, programmable software, and the AI engine array. Vitis serves as a platform development environment, enabling the construction of the graph that connects AI tiles and the development of kernels executed on the AI engines. For verification, throughput calculation, and identification of stream or lock stalls, the Vitis Analyzer is employed. This tool offers detailed insights into the design, aiding in the evaluation and optimization of the implemented algorithms on the ACAP architecture. Throughout this work, we use Vitis and Vivado version 2023.1 running on the AlmaLinux 8.6 Operating System.

To assess the effectiveness of the kernels, requirements are derived from the system specifications provided in the background. As stated before, the requirements originate from the LOFAR system Table 5. While an input bit width of 14 bits should give a sufficient resolution, the AIE only provides a 32-bit floating point datatype. The deployable application must effectively handle streaming data. Samples are initially acquired using ADCs and then read by the programmable logic before being transferred to the AI Engines. Subsequently, the output of the polyphase filter should exit the AI Engines for further processing or to be transported to a central location. We explore the existing AMD DSP libraries in subsection 5.1 containing a potential FIR and FFT instance. In subsection 5.2 we explore custom implementations of the FIR kernels such as the vector processor, pipelining, and parallelization. Finally in subsection 5.3 the individual kernels are combined in a cohesive solution for the Polyphase filter.

Table 5. Input requirements of the LOFAR, Spectrometer, MKID I/II.

	LOFAR 2.0	Spectrometer	A-MKID I/II
Sample rate	200 MSPS	8 GSPS	4.1 GSPS
Input bit width	14 bit	8 bit	12 bit
Receivers per device	96 * 2	1	2
Input data rate	537.6 Gbit/s	64 Gbit/s	196.8 Gbit/s
Branches	512	32k	1024k /2000
Depth	16	8	0/8

5.1 Existing libraries

The initial phase of the design space exploration involves the validation of AIE DSP libraries provided by AMD[42] . The AMD library comprises distinct L1, L2, and L3 primitives, with the current availability limited to L1 and L2. L1 encompasses High-Level Synthesis (HLS) C++ implementations, while L2 consists of AIE C++ implementations. To prevent linking issues, both L1 and L2 primitives are needed in the project. Furthermore, the AMD library offers examples of some L2 implementations, providing a practical reference for developers.

The Vitis tooling offers the functionality to search for existing libraries and integrate them into projects. However, it's worth noting that not all libraries hosted on GitHub are accessible directly from within Vitis. Additionally, the available libraries may not always be up to date or may lack the specific primitives required for execution. As a result, the current best practice entails cloning the necessary GitHub libraries and manually including them in the project.

The verification process of the DSP library's single rate asymmetrical FIR library element "fir_sr_asym_graph" revealed promising initial results for implementing a single FIR filter in the context of the LOFAR application. The straightforward integration involved initializing the FIR as a class and providing the necessary parameters such as data

type and FIR depth. This implementation exhibited favorable performance, requiring only 6 cycles per sample for 16 TAPS. Considering this performance metric, a maximum throughput of 208 Giga samples per second could be achieved when all FIRs are executed in series, aligning well with the LOFAR requirements. Nevertheless, a notable obstacle was encountered while implementing additional FIR filter kernels on a solitary AI engine. The memory allocation designated for the kernel was found to be 512 KiB, theoretically capping the maximum number of FIRs on a single AI tile at 64, thus necessitating 16 tiles for the full deployment of the filterbank. However, during the implementation process, only a maximum of 8 FIR filters were successfully mapped to a single AI engine, despite manually reserving 16 MiB for kernels in the build settings. Presently, the exact limitation remains unclear. as the compiler only indicates that the mapping was unsuccessful. This limitation resulted in the need for 128 AI engines to accommodate a single FIR filter bank, leading to substantial underutilization of the available processing power. Consequently, based on these findings, it becomes evident that the FIR DSP library, as currently configured, may not be well-suited for meeting the specific requirements of the polyphase filter application. The hard limitation of deploying three kernels on a single AIE tile would require a full rewrite of the library making it more efficient to implement the kernel from the ground up. This underscores the need for further exploration and custom software development to better align with the unique demands of the signal processing tasks at hand. The "fir_sr_asym_graph would be very useful when the program necessitates few, very efficient FIRs.

The verification process of the DSP library's FFT kernel, specifically the "fft_ifft_dit_1ch_graph," has yielded promising results. This kernel offers various options for parallelization, including the utilization of the cascade interface to provide partial solutions to neighboring kernels and the ability to distribute spectral points across multiple kernels. The FFT library lacks support for a real FFT. It only supports complex values as input samples. The FFT can still be used with real values by setting the imaginary part to zero however, this necessitates a doubling of the required spectral channels for the LOFAR use case from 512 to 1024 and for the Spectrometer use case from 32k to 64k. During verification, it was determined that an FFT with 1024 spectral points could be deployed on a single AI engine. The throughput of the FFT kernel has demonstrated sufficient performance, reaching an impressive throughput of 220 Mega samples per second without any parallelization enabled and 1024 spectral channels. The FFT faces a constraint in terms of the number of spectral channels, with a hard limit set at 64k. Given these favorable outcomes, it is evident that the DSP FFT kernel is well-suited for the specified use cases of LOFAR, Spectrometer, and A-MKID II, meeting the requirements and providing a robust foundation for the implementation of FFT algorithms on the ACAP architecture. As the A-MKID I needs 1024K spectral channels, the library is not usable for this use case. The performance metrics of the FIR and FFT libraries have been incorporated into the roofline plot shown in Figure 13, facilitating a comparative analysis against the ideal algorithms and serving as a baseline for future custom filter implementations.

5.2 Custom kernel implementations

As previously discussed, the FIR filter bank necessitates the development of custom software to enable the deployment of multiple FIRs on a single AI engine. The upcoming section will detail four kernel implementations, outlining the discoveries and advancements in kernel development for AI engines. The first kernel utilizes the scalar processor and provides a simple implementation of the graph and kernel. The second kernel demonstrates a naive vector processor kernel application as it still partially relies on the scalar processor. The third kernel demonstrates the available vector API, removing the necessity of the scalar engine. Ultimately the fourth kernel culminates in the realization of an efficient and scalable filterbank kernel solution. The results of each implementation are shown in the roofline plot of Figure 13.

5.2.1 Scalar kernel. To construct a baseline, a straightforward FIR filter implementation is devised, leveraging the capabilities of the scalar processor. This approach serves as a foundational step, enabling the clarification of fundamental graph and kernel programming concepts. As reiterated earlier, the deployment of software on AI engines necessitates a combination of dataflow programming, articulated through the graph, and object-oriented programming, embodied in the kernel. Let's first take a look at the graph. Examining the graph in its most elementary form, Listing 1 reveals its core components: it outlines the input and output specifications, initializes the kernel by specifying its source and the proportion of computational resources it utilizes on a single AI engine through the declaration of the runtime ratio. Additionally, the graph establishes the connections between inputs and outputs or between different kernels, providing a foundational framework for orchestrating the flow of data and computations within the signal-processing pipeline.

Listing 1. Core components of an AI engine graph

```
class scalarFIR: public adf::graph {
private:
    kernel fir;
public:
    input_plio  in;
    output_plio out;
    scalarFIR(){

    in  = input_plio::create(plio_32_bits, "data/input.txt");
    out = output_plio::create(plio_32_bits, "data/output.txt");

    fir = kernel::create(scalar_fir);
    source(fir) = "kernels/kernels.cc";
    runtime<ratio>(fir) = 1;

    connect<stream> (in.out[0], fir.in[0]);
    connect<stream> (fir.out[0], out.in[0]);
  }
};
```

The foundational FIR kernel, as illustrated in Listing 2, incorporates a distinction in connection types within the function description, either as a stream or an IO buffer. The stream configuration is tailored for the continuous processing of smaller data packets, featuring a 2x 32-bit input per clock cycle. On the other hand, the IO buffer is optimized for handling larger datasets, with a 2x 256-bit input. The IO buffer operates on a ping-pong concept, wherein the declared size is allocated twice, enabling simultaneous writing to and reading from the buffer. However, a drawback of the IO buffer lies in the necessity to restart the kernel for new data to become accessible, introducing overhead and making it unsuitable for streaming applications. Besides the *readincr()* and *writeincr()* AIE API functions, the remainder of the kernel is implemented in plain C++, enhancing readability and facilitating adaptability. In subsequent stages of the design space exploration, more detailed explanations and optimizations specific to AI engines will be explored.

Listing 2. Scalar fir implementation

```
void scalar_fir(input_stream_float * in, output_stream_float * out) {

        float taps[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        float coeffs[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
        const int SAMPLES = 16;
```

```
        for(int j=0; j<SAMPLES; j++){

                for(int i = sizeof(taps) - 1; i > 0; i--){
                taps[i] = taps[i-1];
                }

                taps[0] = readincr(in);
                int16 output = 0;

                for(int i = 0; i< sizeof(taps); i++){
                        output += coeffs[i] * taps[i];
                }

                writeincr(out, output);
        }
}
```

5.2.2 Vector kernel. The vector kernel involves leveraging the capabilities of the vector processor. This strategic approach is geared towards maximizing theoretical performance by enabling the simultaneous processing of entire vectors within a single clock cycle. The theoretical performance limit, previously computed, incorporates the integration of the vector processor, ensuring a minimum of 2 clock cycles per calculated FIR. Listing 3 showcases the utilization of the vector kernel through the streaming interface. The "chess_prepare_for_pipelining" pragma enables pipelining for this for loop increasing efficiency and throughput. Each loop facilitates the reading of a new sample which is prepended to the vector. The AIE API currently does not support an efficient method to prepend new data to a vector forcing us to write it ourselves using the scalar processor. The vector's historical samples are then subjected to simultaneous multiplication by corresponding coefficients in a singular operation. Subsequently, the partial results undergo summation, a task delegated to the scalar processor. However, this introduces an inherent challenge, as the scalar processor's sequential summation process introduces additional clock cycles for computation. This juxtaposition between the vector processor's parallel capabilities and the scalar processor's sequential summation underscores an aspect warranting further optimization within the computational workflow.

Listing 3. Vector

```
        aie::vector<float, 16> taps = aie::zeros<float, 16>();
        aie::vector<float, 16> coeffs = {aie::load_v<16>(stored_coeffs)};
        aie::accum<accfloat, 16> acc1 = aie::zeros<accfloat, 16>();

        for(int j=0; j<SAMPLES; j++)chess_prepare_for_pipelining{

                for(int i = 15; i > 0; i--){
                taps[i] = taps[i-1];
                }

                taps[0] = readincr(in);
                acc1 = aie::mul(taps, coeffs);
                writeincr(out, aie::reduce_add(acc1.to_vector<float>(0)));
        }
```

5.2.3 Sliding_mul kernel. The subsequent kernel involves the implementation of the *sliding_mull* function, presented in Listing 4, accessible through the aie::vector API. This function facilitates the parallel computation of multiple samples of the same branch, thereby harnessing the vector processor's performance while eliminating the need for the scalar processor in the addition stage. By incorporating this functionality, the computational workflow gains the advantage of enhanced parallelization, contributing to the overall optimization of the process. A throughput of 8 cycles/sample is reached using this algorithm. Despite the substantial increase in potential performance offered by the sliding_mull function, it proves unsuitable for the filterbank. Each branch of the filterbank needs to store 32 history samples in combination with 16 coefficients. This requires at least six AIE to store all the data needed $(4 \text{ Bytes} * (16 * 3) * 1024 \text{ Branches})/32KiB = 6$ AIE making inefficient use of the given resources. A shortcoming is shown as the *readincr()* and *writeincr()* functions are only able to read/write 4 bytes each clock cycle, needing 8 clock cycles to read 8 samples. The *sliding_mul* algorithm needs 4 clock cycles to process the samples and stalls waiting on new data. A solution could be to utilize both incoming stream interfaces, however, this does require the samples to be correctly distributed adding extra complexity. The final and arguably most significant drawback lies in the introduction of batch processing when scaling the algorithm to accommodate multiple branches of the filterbank. The sliding_mul algorithm necessitates waiting for eight samples before starting the calculation. Coupled with the round-robin sampling, the filter must wait for (8 * 1024) samples in total. This segregation between reading samples and processing renders it impossible to hide the processing behind the data transfer, thereby diminishing throughput to 16 cycles/sample.

Listing 4. Implementation of the sliding_mul algorithm

```
// 8 samples
buff.insert(0,readincr_v<8>(sig_in));
acc1 = aie::sliding_mul<8,4>(coefficients1,0,buff,8);
acc2 = aie::sliding_mul<8,4>(coefficients2,0,buff,12);
acc1= aie::sliding_mac<8,4>(acc1, coefficients1,4,buff,16);
acc1= aie::sliding_mac<8,4>(acc2, coefficients2,4,buff,20);
writeincr(sig_out, aie::add(acc1.to_vector<float>(0), acc2.to_vector<float>(0)));
```

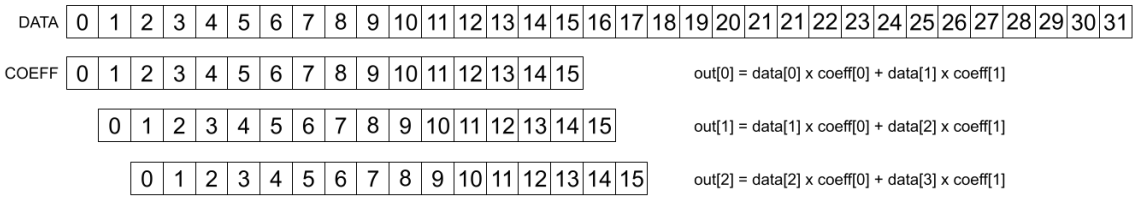


Fig. 11. Visualization of the sliding_mul algorithm

5.2.4 parallel_branched kernel. The last kernel variation that we introduce combines the previously discussed concepts. By simultaneously calculating 8 branches, the entire vector multiply-accumulate path is fully utilized, eliminating the dependency on the scalar processor. However, this optimization brings to light a fundamental limitation of the ACAP, as a single AI engine lacks the storage capacity for the necessary 1024 x 16 history samples (not to mention the 1024 x 16 coefficients) essential for efficient processing. Leveraging the intrinsic functionality of the FIR filter, the design stores only 15 history samples, streaming in the new sample as needed, providing just enough storage for the historical

data. To further address the storage constraint, coefficients are streamed to the kernel, at the cost of introducing extra clock cycles. While the architecture of the AIE should allow to use of neighboring AI engine storage, it was found to be insufficiently implemented in the current tooling, rendering it impractical for this implementation. Figure 12 demonstrates the algorithm requiring 20 clock cycles per sample. Despite the given limitations, the algorithm's ability to exploit parallelism in both branch and depth directions allows for the increase of the overall throughput. Multiple variations of this kernel are designed to allow for parallel and cascaded deployment of kernel instances.

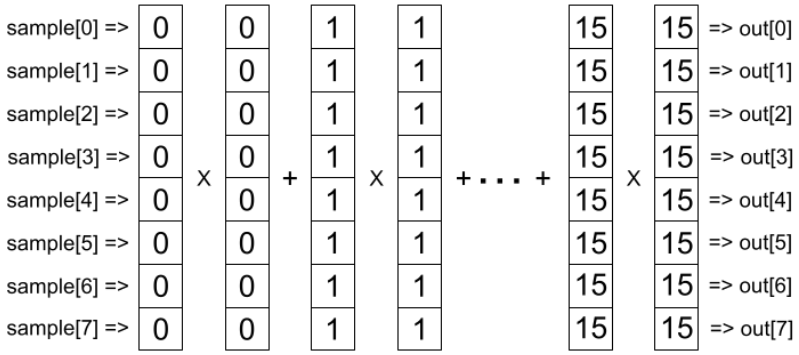


Fig. 12. Visualization of the custom algorithm to process 8 branches in parallel.

Ultimately, five distinct kernels were developed. Both the scalar and vector kernels failed to achieve the required throughput within a reasonable number of AI engines due to their inability to fully utilize the vector processor. Specifically, the scalar kernel attained a throughput of 10 M samples/second per AI engine, while the vector kernel reached a throughput of 40 M samples/second per AI engine. Needing 20 and 5 AI engines to achieve the required throughput respectively. The sliding_mul kernel demonstrated a remarkable throughput of 156 M samples/second per AI Engine when operating with a single branch. However, with the inclusion of additional branches, the efficiency diminishes, resulting in a throughput of 78 M samples/second. Moreover, accommodating multiple branches requires a minimum of six AI engines to store all the data. In conclusion, the parallel kernels achieve a throughput of 63 M samples/second per AI Engine for the streamed variant and 78 M samples/second per AI Engine for the stored variant. Given their superior performance in terms of throughput per AI Engine, these kernels will serve as the foundation for constructing the implementation. Figure 13 shows the kernels plotted on the roofline established in section 4 and Figure 14 gives an overview of throughput compared to the area needed for each kernel.

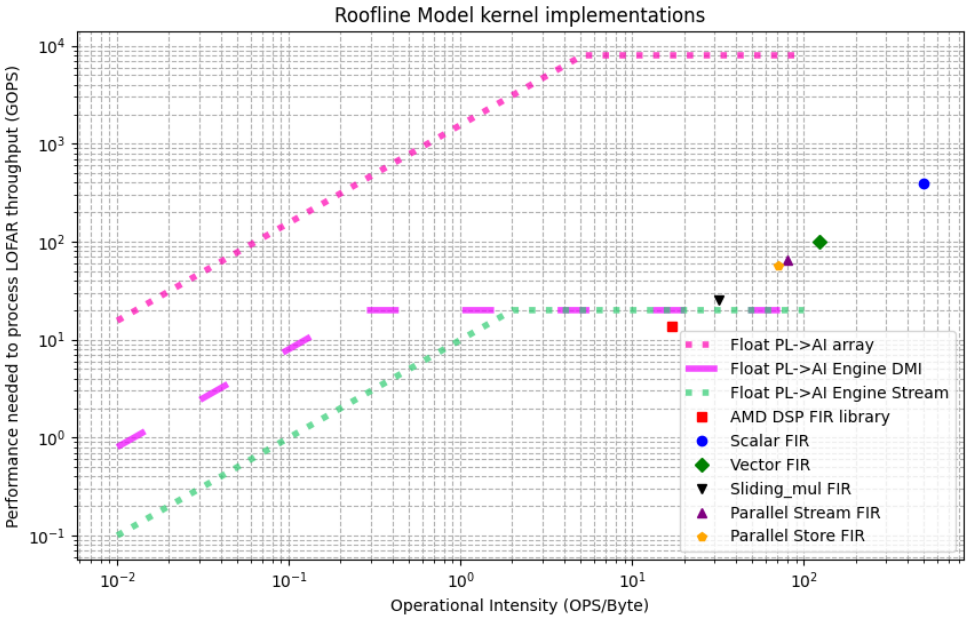


Fig. 13. Roofline plot indicating the effectiveness of the different kernels.

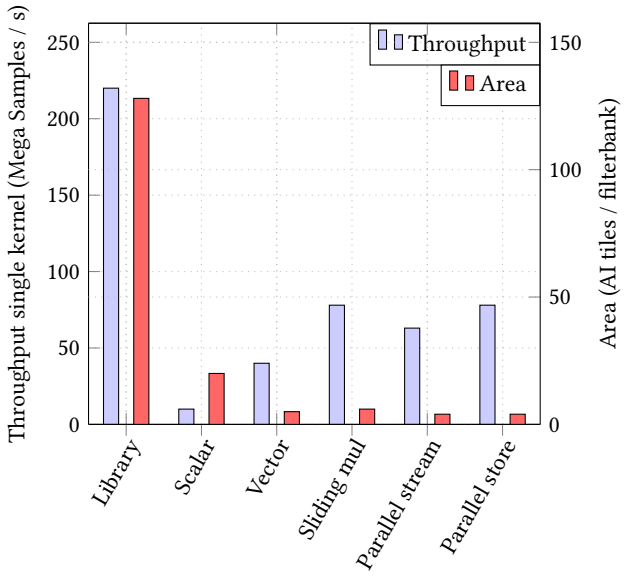


Fig. 14. Bar plot showing the throughput and area of the different kernel implementations

5.3 Dataflow programming

Having established an efficient implementation for the FIR kernel, the focus shifts to dataflow programming, where kernels are connected to form a cohesive program. First, a generic polyphase filter is implemented that should meet the branch and depth requirements of the filter for the LOFAR use case to demonstrate its feasibility. Second the maximum throughput, for the filter, utilizing the entire ACAP array is determined. Finally, a generic polyphase filter library is created able to be tuned to the requirements of other use cases. The envisioned ideal configuration, depicted in Figure 15, entails a singular AI engine for deploying FIR branches and another for executing FFT operations. However, this configuration presents a storage challenge, as mentioned before, requiring 131 KiB of memory for storing coefficients and history samples, surpassing the available 32 KiB of the AI engines local memory.

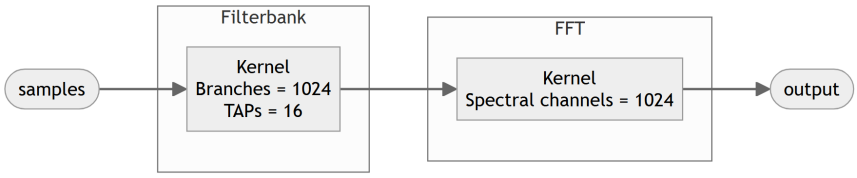


Fig. 15. Ideal dataflow configuration where a single AI Engine is used for the filterbank and a single AI Engine for the FFT.

To address this, parallelization in the branch direction and coefficient streaming is used, splitting the kernel over multiple AI engines as illustrated in Figure 16. This reduces the storage requirement per AI engine to 31 KiB while concurrently enhancing throughput. When using branch parallelization, a single incoming stream needs to be distributed over multiple kernels. In the AI Engine paradigm this is done using packet switching. To manage packet-switching two elements exist, a splitter called *pktsplit* and a merger called *pktmerge*. Packet-switching up to 32 streams is currently supported, so a maximum of 1 to 32 for the splitter and 32 to 1 for the merger. Each packet is prepended with a header called the packed ID indicating the destination. The first branch on the output of the splitter has ID 0, incrementing by 1 for each next kernel. Before the last value of a packet is sent the value "TLAST" is added to indicate the end. The packet-switching splitter element is also shown in Figure 16.

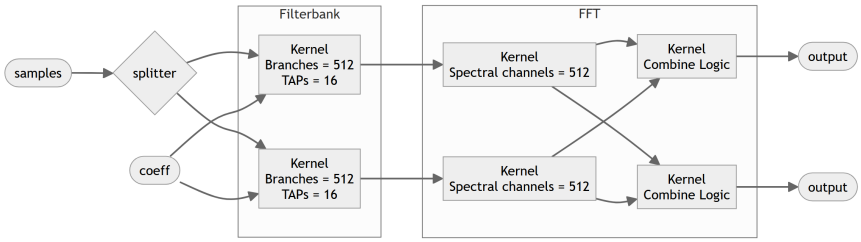


Fig. 16. The dataflow where the filterbank and FFT are both distributed over 2 AI Engines. A splitter is implemented to distribute the samples over the different AI Engines.

An important observation during implementation is the current lack of support for floating-point data types for packet-switching interfaces. This limits possible applications when many different use cases such as in digital signal processing and machine learning rely on the integration of both.
To circumvent this lack of support in the packet-switching interface a solution is implemented for this application

by introducing a kernel that converts the samples from a floating-point to an integer datatype. This introduces extra overhead needing an extra AIE to deploy the kernel taking up valuable resources to circumvent a lack of functionality. Another shortcoming is the lack of native round-robin distribution. Each packet has to be statically addressed to an AIE tile by prepending a header to the data. The addition of this header on the data would need an extra kernel before the implementation of the FIR. Luckily we already have an extra kernel to convert the floating-point samples to integers to which we can add the functionality to prepend a header.

The conversion and prepending of the packet header are executed before the samples reach the splitter, effectively overcoming the issues. It is important to note that while this resolution successfully addresses the datatype and header challenge, it results in an increased number of AI engines to accommodate the additional processing step. The conversion also introduces extra complexity in the parallel_branched kernel as the integer needs to be converted back to floating-point.

Similarly, the absence of functionality for real datatypes in the FFT is addressed by incorporating a kernel that converts real data to imaginary data, a component conveniently provided by AMD. This integration takes place before the FFT operation, effectively resolving the datatype constraint.

To further increase the throughput of the system the cascade interface is used, passing on the partial result and dividing the depth of the FIR filter over multiple AIE tiles. Figure 17 illustrates the entirety of the cohesive dataflow program designed to support a polyphase filter, and Figure 18 shows the linked AI engines in the Vitis analyzer. 10 AI engines are used. to deploy a single instance of the polyphase filter, scaling this to the entire AI engine array would allow for the implementation of 40 polyphase filters on a VC1902 ACAP.

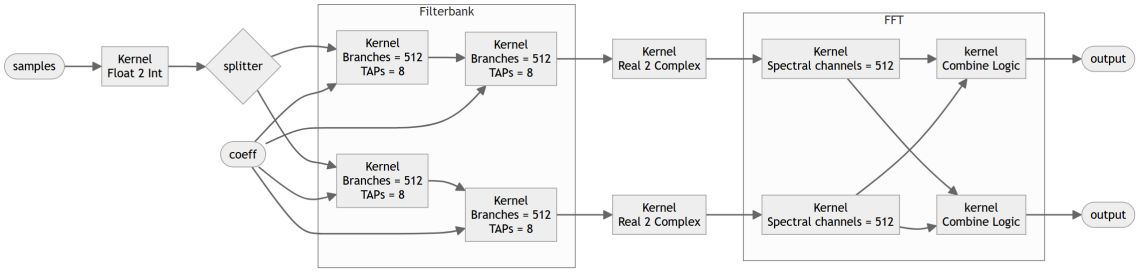


Fig. 17. The Final dataflow implementation for the LOFAR requirements includes a Float2Int kernel, splitter, 4x filterbank, Real2complex kernels and FFT.

In exploring the limits of the ACAP in another direction, we use the same implementation of the filter for the LOFAR use case, but this time optimize a single filter for maximum throughput. The dataflow implementation of the polyphase filter uses 384 AIE tiles, with 256 AIE tiles for the FIR, 32 AI engines in parallel, and 8 in series. An additional 32 tiles handle data conversion from real to complex, and 96 tiles are dedicated to the FFT process. This setup allows for a throughput of 16 Giga Samples Per Second, showcasing the system's capability to handle high-speed data effectively. When structuring a dataflow layout encompasses a large area, the Vitis tooling encounters difficulties in managing layouts and connecting kernels. Despite offering the option to constrain a graph or subgraph to a specific location, the tool occasionally times out during layouting even when this feature is utilized.

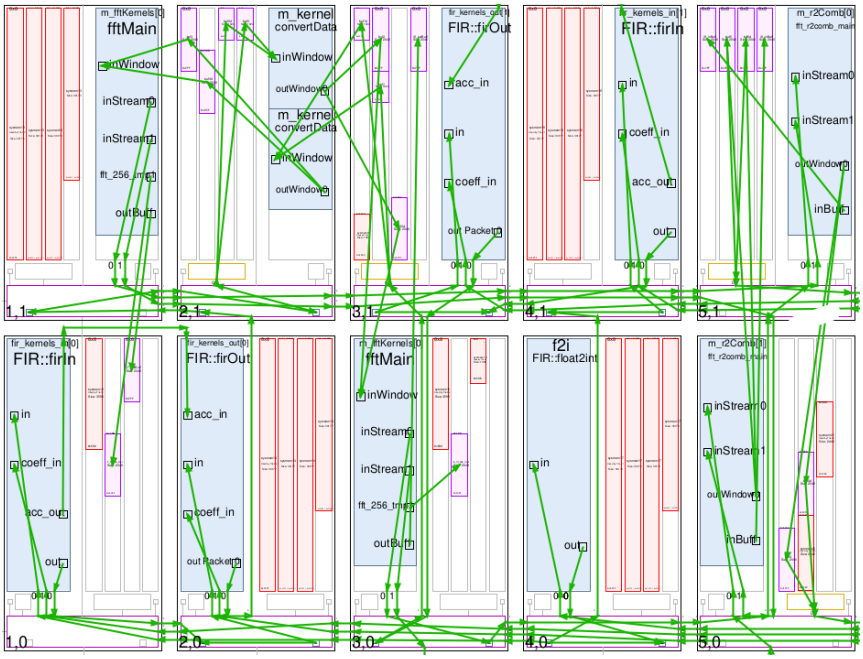


Fig. 18. The implementation of the polyphase filter as shown in the Vitis Analyzer

5.4 Polyphase filter library

With a functional dataflow meeting the requirements for the LOFAR use case, we extend these findings to a generic polyphase filter library implementation. This library allows users to create a custom polyphase filter layout specific to their needs. By using the heuristics below the user can determine the layout needed for their application and supply a top-level graph with the necessary information.

When applying the developed polyphase filter to other use cases, five elements impact the dataflow layout:

- The required throughput in samples/s
- The sample datatype
- The number of FIR branches
- The depth of the FIR
- The need/want to store coefficients

We use a heuristic to automatically indicate the dataflow layout indicated in the flowchart of Figure 19. It separates the two cases where the coefficients are stored or streamed. In each case the amount of kernels is calculated, needed to support the incoming samples, store the sample data, and reach the minimal throughput. The greatest value indicates the minimal number of kernels needed overall.

To determine the layout of these kernels, three rules should be followed. First, the branches should be evenly distributed over the parallel kernels ensuring that samples are distributed to the correct branch. Second, we ensure that the TAPs are evenly distributed among the cascaded kernels. The number of parallel AI Engines used for the FFT should be equal to the parallel AIEs used for the filterbank.

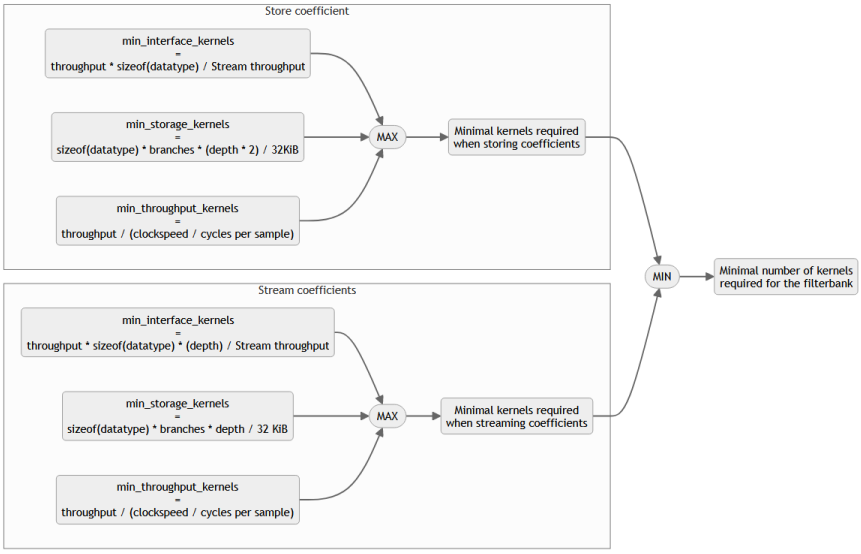


Fig. 19. Heuristic to determine the minimal number of kernels needed to get a valid filterbank dataflow layout.

- branches >= min_interface_kernels value of the store coefficients
- branches % parallel_kernels == 0
- depth % cascaded_kernels == 0
- parallel_kernels == parallel_aie_FFT

This heuristic can be applied to the use cases discussed in the background. A-MKID I lacks a filter bank and directly routes incoming samples to the FFT. However, with 1024k spectral channels, it faces a challenge. The current FFT library from AMD supports a maximum of 64k spectral channels, rendering it unable to meet the A-MKID I requirement of 1024k spectral channels at the time of writing. For the A-MKID I requirements to be met, a new FFT should be designed, not limited to 64 spectral channels.

In contrast to A-MKID I, A-MKID II incorporates a filter bank with fewer spectral channels, allowing it to meet the FFT requirements. The high sample rate necessitates a minimum of 4 parallel branched AIE tiles when storing the coefficients following from the data-rate and datatype 4.1 G samples/s $*$ 4 (bytes/sample) $=$ 16.4 GiB/s. Given the throughput of a single streaming interface 16.4 GiB/s/5 GiB/s $=$ 4 AIE. The high sample rate makes streaming coefficients to the AI engines inefficient, requiring at least 17 engines for throughput $(16.4\ \text{GiB/s} * 4)/5\ \text{GiB/s} = 14$ AIE. The combination of the relatively low number of branches and low depth allows for the storage of all history samples on a single AI engine when streaming $(4 * 2000 * 4)/32768 = 0.98$ AIE and 2 AIE when storing the coefficients. The assumption is made that a single branch can be calculated in 5 clock cycles, due to the depth being 1/4 of the depth of the filterbank for the LOFAR use case. This allows for the calculation of the minimum number of AIE tiles needed to achieve the required overall throughput: $\frac{4.1\ \text{Gsamples/s}}{(1.25\ \text{GHz}/5)} = 17$ AIE. In the end, the number of AIE needed to achieve the requirements of the A-MKID II is not bound by whether the coefficients are streamed or stored.Figure 20 shows the heuristic applied to the A-MKID I requirements.

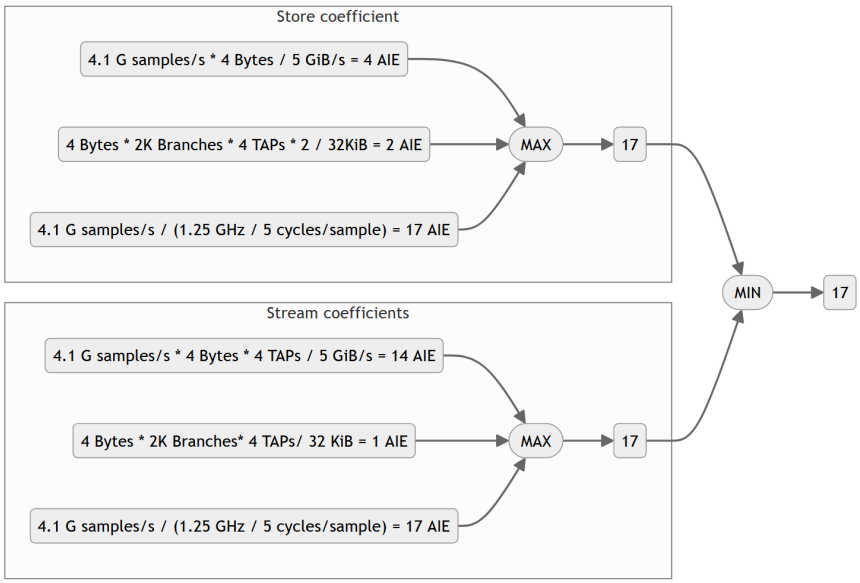


Fig. 20. The heuristics applied to the MKID I requirements

The Spectrometer exhibits a sample throughput of 8 G samples/s, resulting in a total throughput of 32 GiB/s, necessitating a minimum of 7 parallel AI tiles. Simultaneously, its coefficient throughput, calculated at 32 GiB/s $*$ 4 = 128 GiB/s, requires a minimum distribution across 26 AIE tiles to prevent stalling. Storing the history samples involves allocating a minimum of 32 AIE tiles to manage 64k branches $(4 * 64000 * 4)/32$KiB. The custom FIR filter's ability to compute a sample every 5 clock cycles is again attributed to its depth being a factor 4 smaller than 16. Reaching a maximum throughput of 8 G samples/s when accounting for the minimum of 32 AI Engines tiles. From this, a possible configuration is found with 16 parallel kernels and 2 cascaded kernels. It is noted that these requirements test the limits of the ACAP as it reaches the limit of 64000 spectral channels of the FFT library. Figure 21 shows the heuristic applied to the A-MKID I requirements.

The created library shown as a class diagram in Figure 22 can be included in a project where the top-level graph *PolyphaseFilter* is called and initialized. The user would still need to connect the polyphase library to input and outputs.

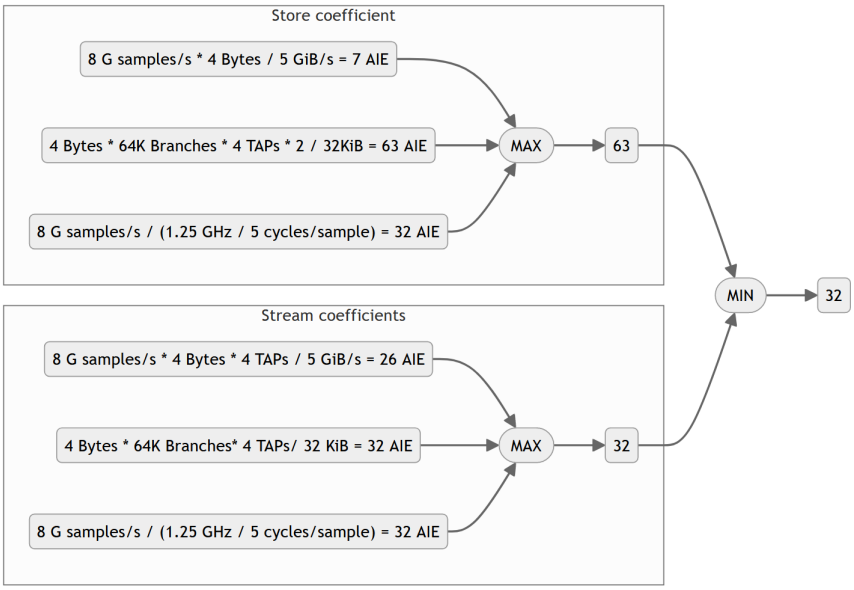


Fig. 21. The heuristics applied to the Spectrometer requirements

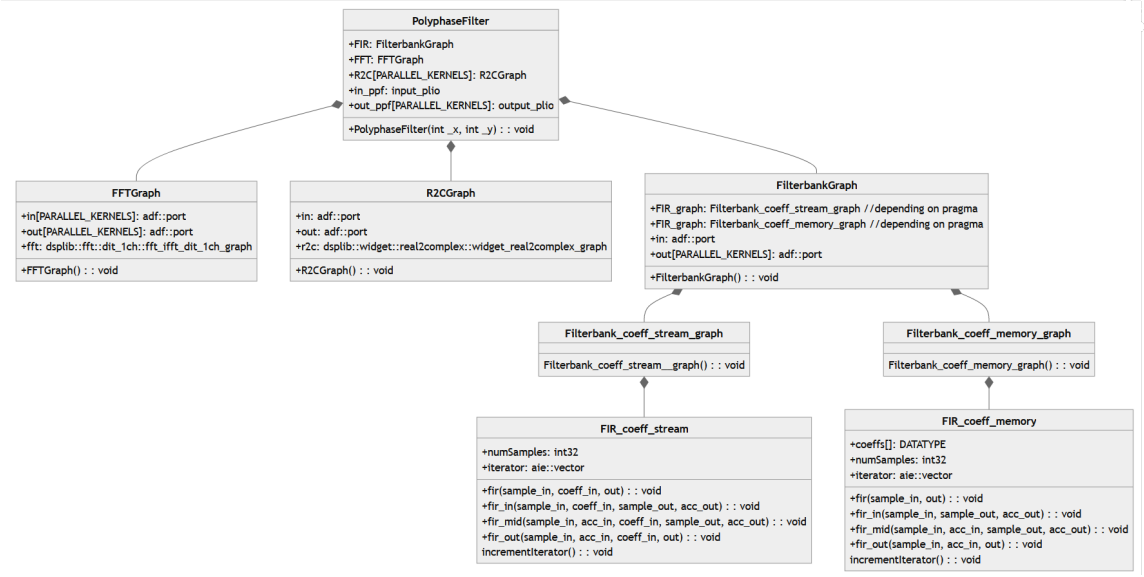


Fig. 22. The class structure of the created Polyphase library.

5.5 AIE-ML

AMD has, more recently, introduced a promising variant of the AI engine known as AIE-ML [51]. This version exhibits significant enhancements, including a doubling of the local data memory per tile to 64 kB. Additionally, the introduction

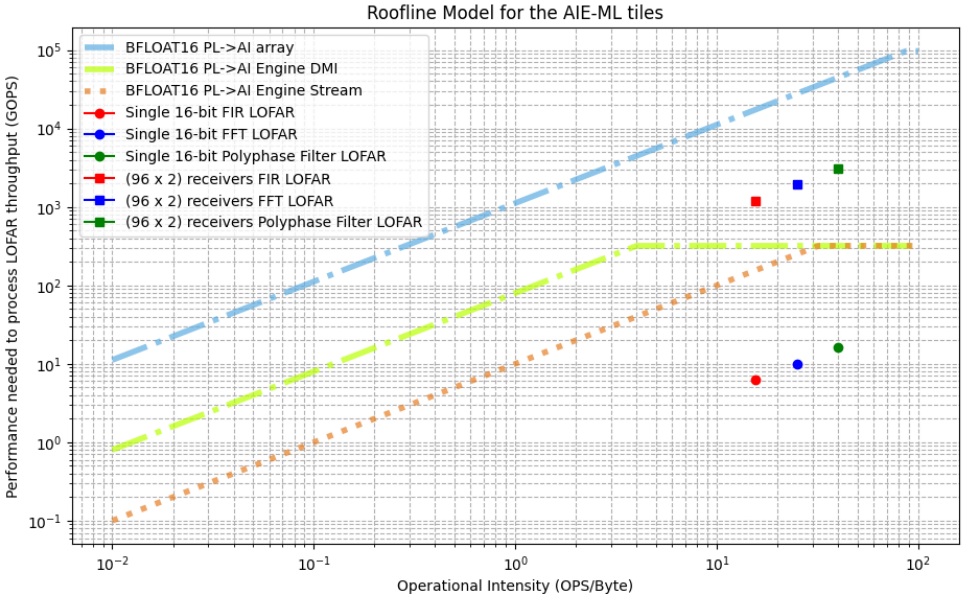


Fig. 23. The roofline plot of the AIE-ML ACAP. When compared to the previous rooflines, the horizontal lines are substantially higher.

of the bfloat16 datatype effectively doubles the available interface throughput and storage capacity. Notably, the bfloat16 datatype offers a remarkable 16x increase in compute performance compared to the 32-bit floating point, delivering an impressive 256 operations/cycle.

The combination of doubling available memory, reducing datatype size, and increasing compute capacity enables the storage of all history samples and coefficients on a single AI engine, alleviating the main limitation of the first generation AIE. The increase in available compute results in a 16x increase in theoretical maximum throughput, theoretically meeting the requirements of the LOFAR use case on a single device.

When mapping the optimal algorithms identified in section 4 onto the roofline plot generated using the available compute resources on AIE-ML, as illustrated in Figure 23, the plotted points appear in the same location. However, the roofline itself is shifted upwards. These enhancements could allow the realization of the ideal dataflow graph depicted in Figure 15, reducing complexity and increasing scalability.

6 CONCLUSION & FUTURE WORK

In conclusion, the theoretical analysis conducted on the polyphase filter implementation on the ACAP platform reveals intriguing prospects for its utilization. Notably, a single AIE tile theoretically contains enough computational capability to deploy a complete polyphase filter, while the entire AIE array could potentially support an entire LOFAR station.

Evaluation of existing AI engine libraries provided by AMD sheds light on their applicability to our use case. While the FFT library presents functionality limitations, particularly in supporting only complex values and restricting spectral channels to 64k, hindering the realization of A-MKID I requirements it applies to the other systems. The FIR library emerges as remarkably efficient, boasting a throughput of $2.1 * 10^8$ samples/s on a single AI engine when implemented with 16 TAPs. However, challenges arise in effectively parallelizing FIR kernels, rendering them unsuitable for these applications. The implementation of a real-complex FFT or an FFT not limited to 64k spectral channels would be a great asset in the future.

Custom FIR kernels leveraging various features of the AI engines have been developed, with the final iteration achieving a throughput of $7.8 * 10^7$ samples/s on a single AIE using 16 TAPs.

The current state of tooling underscores the need for further refinement, particularly in addressing deficiencies such as the absence of floating-point support when using packet-switching, a critical component of our system and other AI applications.

Despite these challenges, the development of a versatile library capable of accommodating a wide range of requirements and enabling effective parallelization in both branch and depth directions is a significant milestone. Notably, the current system demonstrates feasibility in meeting the demanding specifications of the LOFAR, Spectrometer, and A-MKID II use cases.

Despite promising advancements such as the release of AIE-ML with enhanced performance and a 16-bit bfloat, the platform's maturity of tooling remains a significant hurdle, as available features can not be optimally utilized. Addressing these challenges is crucial to fully harness the potential of the ACAP platform, particularly in accommodating the use cases of the LOFAR, Spectrometer, and A-MKID II systems.

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon Europe research and innovation program under grant agreement No 101093934, and from the Netherlands eScience Center through the RECRUIT project. The author would like to thank Vincent Sprave, Daniele Passaretti, and Thilo Pionteck from the Otto-von-Guericke-Uinversitat Mageburg for collaboration on the ACAP architecture. Gerrit Grutzeck from the MAX-PLANCK-INSTITUT for Radioastronomy for supplying use cases for the Spectrometer and MKID and his expertise with regards to radio telescopes. Friends and Family for the kind words and for hearing me ramble on about polyphase filters the past year. Finally, above all the author would like to thank Steven van der Vlugt for his invaluable guidance and great collaboration during the project.

REFERENCES

[1] A. R. Thompson, J. M. Moran, et al. "Interferometry and Synthesis in Radio Astronomy." 2017. doi:10.1007/978-3-319-44431-4. URL http://link.springer.com/10.1007/978-3-319-44431-4.

[2] M. P. Van Haarlem, M. W. Wise, et al. "LOFAR: The LOw-Frequency ARray." *Astronomy & Astrophysics*, volume 556:p. A2, 8 2013. ISSN 0004-6361. doi:10.1051/0004-6361/201220873. URL https://www.aanda.org/articles/aa/full_html/2013/08/aa20873-12/aa20873-12.htmlhttps://www.aanda.org/articles/aa/abs/2013/08/aa20873-12/aa20873-12.html.

[3] S. Heyminck, U. U. Graf, et al. "GREAT: The SOFIA high-frequency heterodyne instrument." *Astronomy and Astrophysics*, volume 542, 2012. ISSN 00046361. doi:10.1051/0004-6361/201218811. URL https://www.researchgate.net/publication/221679697_GREAT_the_SOFIA_high-frequency_heterodyne_instrument.

[4] D. I. Kaplum, D. M. Klionskiy, et al. "Application of polyphase filter banks to wideband monitoring tasks." *Proceedings of the 2014 IEEE North West Russia Young Researchers in Electrical and Electronic Engineering Conference, ElConRusNW 2014*, pp. 95–98, 2014. doi:10.1109/ELCONRUSNW.2014.6839211.

[5] F. J. Harris, C. Dick, et al. "Digital receivers and transmitters using polyphase filter banks for wireless communications." *IEEE Transactions on Microwave Theory and Techniques*, volume 51(4 II):pp. 1395–1412, 4 2003. ISSN 00189480. doi:10.1109/TMTT.2003.809176.

[6] C. Y. Chou and C. Y. Wu. "The design of wideband and low-power CMOS active polyphase filter and its application in RF double-quadrature receivers." *IEEE Transactions on Circuits and Systems I: Regular Papers*, volume 52(5):pp. 825–833, 2005. ISSN 10577122. doi:10.1109/TCSI.2005.846672.

[7] B. B. Cheng, J. Xu, et al. "The application of poly-phase filter in rader signal processing." *2010 6th International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2010*, 2010. doi:10.1109/WICOM.2010.5600615.

[8] B. Klein, S. D. Philipp, et al. "The APEX digital Fast Fourier Transform Spectrometer." *A&A*, volume 454:pp. 29–32, 2006. doi:10.1051/0004-6361:20065415. URL http://dx.doi.org/10.1051/0004-6361:20065415.

[9] B. Klein, S. Hochgürtel, et al. "High-resolution wide-band fast Fourier transform spectrometers." *Astronomy and Astrophysics*, volume 542:p. L3, 2012. ISSN 00046361. doi:10.1051/0004-6361/201218864. URL https://ui.adsabs.harvard.edu/abs/2012A&A...542L...3K/abstract.

[10] Max Planck Institute for Radio Atronomy. "A-MKID | Max Planck Institute for Radio Astronomy." URL https://www.mpifr-bonn.mpg.de/4489022/a-mkid.

[11] H. Ramon, H. Li, et al. "Efficient Parallelization of Polyphase Arbitrary Resampling FIR Filters for High-Speed Applications." *Journal of Signal Processing Systems*, volume 90(3):pp. 295–303, 3 2018. ISSN 19398115. doi:10.1007/S11265-017-1235-9/TABLES/4. URL https://link.springer.com/article/10.1007/s11265-017-1235-9.

[12] C. N. Ang, R. H. Turner, et al. "Virtex FPGA implementation of a polyphase filter for sample rate conversion." *Conference Record of the Asilomar Conference on Signals, Systems and Computers*, volume 1:pp. 365–369, 2000. ISSN 10586393. doi:10.1109/ACSSC.2000.910979.

[13] P. Fiala and R. Linhart. "High performance polyphase FIR filter structures in VHDL language for Software Defined Radio based on FPGA." In "2014 International Conference on Applied Electronics," volume 2015-January, pp. 83–86. IEEE, 9 2014. ISBN 978-8-0261-0277-9. doi:10.1109/AE.2014.7011674. URL http://ieeexplore.ieee.org/document/7011674/.

[14] K. Van Der Veldt, R. Van Nieuwpoort, et al. "A polyphase filter for GPUs and multi-core processors." *Astro-HPC '12 - Workshop on High-Performance Computing for Astronomy*, pp. 33–40, 2012. doi:10.1145/2286976.2286986. URL https://dl.acm.org/doi/10.1145/2286976.2286986.

[15] K. Adámek, J. Novotný, et al. "A polyphase filter for many-core architectures." *Astronomy and Computing*, volume 16:pp. 1–16, 11 2015. doi:10.1016/j.ascom.2016.03.003. URL http://arxiv.org/abs/1511.03599http://dx.doi.org/10.1016/j.ascom.2016.03.003.

[16] P. Butler and A. Devices. "Antialiasing Filtering Considerations for High Precision SAR Analog-to-Digital Converters."

[17] C. Zhang, T. Geng, et al. "H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture." *Proceedings - 2022 32nd International Conference on Field-Programmable Logic and Applications, FPL 2022*, pp. 200–208, 2022. doi:10.1109/FPL57034.2022.00040.

[18] A. Al-Zoubi, G. Martino, et al. "CNN Implementation and Analysis on Xilinx Versal ACAP at European XFEL." *International System on Chip Conference*, volume 2022-September, 2022. ISSN 21641706. doi:10.1109/SOCC56010.2022.9908101.

[19] T. Zhang, D. Li, et al. "A-U3D: A Unified 2D/3D CNN Accelerator on the Versal Platform for Disparity Estimation." *Proceedings - 2022 32nd International Conference on Field-Programmable Logic and Applications, FPL 2022*, pp. 123–129, 2022. doi:10.1109/FPL57034.2022.00029.

[20] X. Jia, Y. Zhang, et al. "XVDPU: A High Performance CNN Accelerator on the Versal Platform Powered by the AI Engine." *Proceedings - 2022 32nd International Conference on Field-Programmable Logic and Applications, FPL 2022*, pp. 209–217, 2022. doi:10.1109/FPL57034.2022.00041.

[21] W. Zhang, T. Wang, et al. "New Filter2D Accelerator on the Versal Platform Powered by the AI Engine." pp. 437–449. Springer, Singapore, 2024. doi:10.1007/978-981-99-7872-4{_}24. URL https://link.springer.com/10.1007/978-981-99-7872-4_24.

[22] J. Zhuang, J. Lau, et al. "CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture." *FPGA 2023 - Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 153–164, 1 2023. doi:10.1145/3543622.3573210. URL https://arxiv.org/abs/2301.02359v1.

[23] Y. Xiao and Y. Liang. "Flexible Acceleration Framework for Dense/Sparse Matrix Multiplication on Versal ACAP." pp. 01–02, 8 2023. doi:10.1109/ISEDA59274.2023.10218584.

[24] J. Zhuang, Z. Yang, et al. "High Performance, Low Power Matrix Multiply Design on ACAP: from Architecture, Design Challenges and DSE Perspectives." pp. 1–6, 9 2023. ISSN 0738100X. doi:10.1109/DAC56929.2023.10247981.

[25] C. Broekema. "Commodity compute and data-transport system design in modern large-scale distributed radio telescopes." *PhD thesis, Vrije Universiteit Amsterdam*, 2020.

[26] "Westerbork Synthesis Radio Telescope - Wikipedia." URL https://en.wikipedia.org/wiki/Westerbork_Synthesis_Radio_Telescope.

[27] R. L. Brown, W. Wild, et al. "ALMA – the Atacama large millimeter array." *Advances in Space Research*, volume 34(3):pp. 555–559, 1 2004. ISSN 02731177. doi:10.1016/j.asr.2003.03.028. URL https://linkinghub.elsevier.com/retrieve/pii/S0273117703011979.

[28] D. Schaubert, A. van Ardenne, et al. "The square kilometer array (ska) antenna." In "IEEE International Symposium on Phased Array Systems and Technology, 2003.", volume 2003-January, pp. 351–358. IEEE, 2003. ISBN 0-7803-7827-X. doi:10.1109/PAST.2003.1257007. URL http://ieeexplore.ieee.org/document/1257007/.

[29] U. U. Graf, I. Barrueto, et al. "The CCAT-prime Heterodyne Array Instrument (CHAI)." *pcsf*, p. 339, 2023. URL https://ui.adsabs.harvard.edu/abs/2023pcsf.conf..339G/abstract.

[30] Analog Dialogue. "Phased Array Beamforming ICs Simplify Antenna Design | Analog Devices.", 2019. URL https://www.analog.com/en/analog-dialogue/articles/phased-array-beamforming-ics-simplify-antenna-design.html.

[31] D. C. Price. "Chapter 1 Spectrometers and Polyphase Filterbanks in Radio Astronomy." *The WSPC Handbook of Astronomical Instrumentation: Volume 1: Radio Astronomical Instrumentation*, pp. 159–179, 2021.

[32] EPCC. "VCK5000 Architecture." URL https://fpga.epcc.ed.ac.uk/docs/vck5000_building_emulation.html.

[33] "AI Engine Technology." URL https://www.xilinx.com/products/technology/ai-engine.html.

[34] "AI Engine Interfaces Versal Adaptive SoC AI Engine Architecture Manual (AM009) Reader AMD Adaptive Computing Documentation Portal." URL https://docs.xilinx.com/r/en-US/am009-versal-ai-engine/AI-Engine-Interfaces.

[35] "Functional Overview Versal Adaptive SoC AI Engine Architecture Manual (AM009) Reader AMD Adaptive Computing Documentation Portal." URL https://docs.xilinx.com/r/en-US/am009-versal-ai-engine/Functional-Overview.

[36] J. P. Smith, J. I. Bailey, et al. "A High-Throughput Oversampled Polyphase Filter Bank Using Vivado HLS and PYNQ on a RFSoC." *IEEE Open Journal of Circuits and Systems*, volume 2:pp. 241–252, 1 2021. ISSN 2644-1225. doi:10.1109/OJCAS.2020.3041208. URL https://ieeexplore.ieee.org/document/9336352/.

[37] L. H. Arnaldi. "Implementation of a Polyphase Filter Bank Channelizer on a Zynq FPGA." *2020 Argentine Conference on Electronics, CAE 2020*, pp. 57–62, 2 2020. doi:10.1109/CAE48787.2020.9046377.

[38] B. Merry. "Efficient channelization on a Graphics Processing Unit." *Journal of Astronomical Telescopes, Instruments, and Systems*, volume 9(03), 3 2023. doi:10.1117/1.JATIS.9.3.038001. URL http://arxiv.org/abs/2303.09886http://dx.doi.org/10.1117/1.JATIS.9.3.038001.

[39] N. Ragoomundun and G. K. Beeharry. "A hybrid SDR-GPU receiver for a low-frequency array in radio astronomy." *Experimental Astronomy*, volume 47(3):pp. 313–324, 6 2019. ISSN 15729508. doi:10.1007/S10686-019-09629-9/FIGURES/6. URL https://link.springer.com/article/10.1007/s10686-019-09629-9.

[40] N. Perryman, C. Wilson, et al. "Evaluation of Xilinx Versal Architecture for Next-Gen Edge Computing in Space." *IEEE Aerospace Conference Proceedings*, volume 2023-March, 2023. ISSN 1095323X. doi:10.1109/AERO55745.2023.10115906.

[41] F. Johansson and L. Ögnelod. "Beamforming of Multi-Channel Digital Radar System on System-on-Chip." 2022.

[42] "Vitis_Libraries/dsp at main · Xilinx/Vitis_Libraries · GitHub." URL https://github.com/Xilinx/Vitis_Libraries/tree/main/dsp.

[43] AMD. "Xilinx FIR Tutorial.", 2021. URL https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/AI_Engine_Development/Design_Tutorials/07-firFilter_AIEvsHLS/README.html#revision-history.

[44] AMD. "Xilinx FFT Tutorial.", 2021. URL https://xilinx.github.io/Vitis-Tutorials/2021-2/build/html/docs/AI_Engine_Development/Design_Tutorials/06-fft2d_AIEvsHLS/README.html.

[45] "Overview AI Engine Kernel and Graph Programming Guide (UG1079) Reader AMD Adaptive Computing Documentation Portal." URL https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding/Overview?tocId=Bk5qNxVt~yrjda~iVyOTeQ.

[46] "Overview Versal ACAP AI Engine Programming Environment User Guide (UG1076) Reader AMD Adaptive Computing Documentation Portal." URL https://docs.xilinx.com/r/2021.2-English/ug1076-ai-engine-environment/Overview.

[47] "AI Engine Intrinsics." URL https://www.xilinx.com/htmldocs/xilinx2021_2/aiengine_intrinsics/intrinsics/index.html.

[48] "AI Engine API User Guide." URL https://www.xilinx.com/htmldocs/xilinx2022_2/aiengine_api/aie_api/doc/index.html.

[49] S. Williams, A. Waterman, et al. "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*." 2008.

[50] D. Miles. "Compute Intensity and the FFT." *Proceedings of the 1993 ACM/IEEE conference on Supercomputing,*, 1993.

[51] "AIE-ML Processor • Versal Adaptive SoC AIE-ML Architecture Manual (AM020) • Reader • AMD Adaptive Computing Documentation Portal." URL https://docs.xilinx.com/r/en-US/am020-versal-aie-ml/AIE-ML-Processor.

ACRONYMS

ACAP Adaptive Compute Acceleration Platform. 1, 2, 6, 27

ADC Anolog to Digital Converter. 3

AIE AI Engines. 5–7, 9, 14, 16, 17, 20, 24–27, 29

CNN Convolutional Neural Network. 2

DMI Data Memory Interface. 7

FIR Finite Impulse Response. 5
FPGA Field Programmable Gate Arrays. 2, 4, 6

GCN Graph Convolutional Network. 2
GOPS Giga Operations Per Seconds. 9, 13
GSPS Giga Samples Per Second. 24

HBA High-Band Antenna. 3, 4

LBA Low-Band Antenna. 3, 4

MKID Microwave Kinetic Inductance Detector. 2, 3
MSPS Mega Samples Per Second. 4

NOC Network on Chip. 7

OPS Operations Per Second. 13

PL Programmable Logic. 6, 7
PS Programmable Subsystems. 6, 7

SIMD Single Instruction Multiple Data. 9

VLIW Very Long Instruction Words. 7, 9