Universiteit Leiden
The Netherlands

Opleiding Informatica

An Optimized Polyphase Filter Bank

for NVDLA

H. Lambert

Supervisors:
Prof.dr. R.V. van Nieuwpoort
Dr. J.W. Romein

BACHELOR THESIS

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

09/07/2025

Abstract

Radio astronomy pipelines, such as that of the LOFAR telescope network, require high-throughput and energy efficient signal processing. Although GPUs are commonly used for acceleration of these pipelines, this paper investigates the effectiveness of using NVIDIA NVDLA, a deep learning accelerator, for this task. Specifically, it focuses on a polyphase filter bank (PFB). We show how a PFB can be mapped to neural network operations, which can be executed on the NVDLA with 16-bit floating point (FP16) precision. For a PFB based on an FFT, we report a maximum throughput on the order of 10^4 samples/s and for a PFB based on a DFT 10^6 samples/s. The reference GPU implementation shows a throughput of 10^9 samples/s. Therefore, we conclude that a PFB on NVDLA cannot compete with a CUDA implementation with FP16 precision.

Acknowledgements

I would like to give special thanks to my supervisors, Prof.dr R.V. van Nieuwpoort and Dr. J.W. Romein for their valuable feedback and guidance throughout this thesis.

This work was funded by the EU HORIZON INFRA-TECH (RADIOBLOCKS, Grant Agreement nr. 101093934)

Contents

1 Introduction

In the field of radio astronomy, the cosmos is not observed through the visible part of electromagnetic radiation (light), but instead through radio waves, which are the lower frequencies of electromagnetic radiation. Radio waves are emitted by various processes throughout the universe and detecting these waves helps us to gain new insights into how the universe evolves.

The detection of such radio waves is mostly accomplished with antennas instead of traditional telescopes used for visible light astronomy. Most radio telescopes consist of multiple antennas, combining these antennas together increases sensitivity or allows for more frequent sampling with the same sensitivity. Because radio telescopes can consist of a large number of antennas, the amount of data that they generate can be hard to process, not only computationally, but also in the amount of power it requires.

LOFAR is such a radio telescope network, of which the core is located in the Netherlands. LOFAR currently does beamforming at the stations, which greatly reduces the data rate. But most of the further processing is done centrally in Groningen, which limits the flexibility of per station processing, such as noise filtering with a local sensor. Moving more of the LOFAR processing pipeline to the stations would enable greater flexibility for such features and make them easier to implement and scale.

Historically, throughput and power requirements of performing too much of the pipeline on location at each station were limiting factors. Hardware, however, has become much more power efficient and can handle much higher throughputs than before. The use of GPUs to accelerate the pipeline has already been studied [1]. GPUs are already more power efficient for easy to parallelize operations than CPUs, but still require a substantial amount of power. This is where NVIDIA Jetson comes in. The NVIDIA Jetson line is designed for edge computing and offers great performance at low power usage.

This thesis aims to investigate whether part of the LOFAR pipeline can be implemented efficiently on the NVIDIA Jetson line. More specifically, on NVDLA. NVDLA is an energy efficient, deep learning accelerator that is designed to run inferences of neural networks. It shares the memory of the CPU and GPU and in the Jetson Orin AGX, it accounts for 105 out of the 275 INT8 tops [2]. Moving processing from the GPU to NVDLA leaves the GPU to do other tasks.

The part of the pipeline that will be explored is the polyphase filter bank. A polyphase filter bank can be seen as a prism in software that splits a wider band of samples into narrower sub-bands, trading time resolution for frequency resolution. The effectiveness will be investigated by researching the following main question: *How does a polyphase filter implemented in NVDLA perform compared to a CUDA implementation?* which in turn can be answered by the following sub-questions. Firstly: *What is the difference in throughput between the NVDLA and a CUDA kernel?* Secondly: *What is the difference in throughput per joule between the CUDA kernel and NVDLA?* And lastly: *What is the difference in accuracy between a CUDA kernel and NVDLA?*

The implementations and benchmarking code discussed throughout this thesis are available in an accompanying repository on GitHub [1].

[1] https://github.com/HendrikLambert/nvdla-pfb

2 Background

In this section, we will explain the required background information. This includes a section about radio astronomy, an overview of LOFAR, a technical description of a polyphase filter bank and an overview of how the hardware and software of NVDLA works.

2.1 Radio Astronomy

As stated above, radio astronomy is the subfield of astronomy that focuses on radio waves. After Maxwell devised his equations for electromagnetic radiation in the 1860s. People started theorizing about waves outside the visible spectrum coming from space. In the 1930s radio waves were first detected, after this the science community started to build more and bigger radio telescopes.

There are several difficulties to overcome when making modern radio telescopes that keep producing better results than previous generations. One big problem is interference. Since radio waves from the universe are very faint, even small amounts of interference can make the real signal indistinguishable. Devices like radios, GPS, cars and so many more operate on the same frequencies as radio telescopes detect. This interference is made even noisier by the reflections from planes or wind turbines. Combined, this makes some frequencies, or time slots, so distorted that it cannot be used for further analysis.

A second problem with radio telescopes is that radio waves have longer wavelengths/lower frequencies than visible light, which are detected by optical telescopes. The maximum theoretical angular resolution of a telescope can be described with equation (2.1) [3, p. 150].

$$\theta_{min} = 1.22\frac{\lambda}{D} \qquad \text{(Assuming } |\theta| \ll 1\text{)} \tag{2.1}$$

$$\lambda = \frac{c}{f} \tag{2.2}$$

Combine equation (2.1) with the equation for wavelength (2.2), assuming a vacuum, gives:

$$\theta_{min} = 1.22\frac{1}{D}\frac{c}{f} \Rightarrow \theta_{min} \propto \frac{1}{fD} \tag{2.3}$$

Where θ_{min} equals the minimum angular resolution that one can detect in radians, λ the wavelength detected, D the diameter of the lens aperture, f the frequency and c the speed of light in a vacuum.

Equation (2.3) describes the relation between the resolution of a telescope and the frequency multiplied by the diameter of the telescopes. Red light (the lowest frequency of visible light) has a wavelength of 650 nm $\approx 4.62 \cdot 10^{14}$ Hz. The highest frequency LOFAR can detect is 240 Mhz, which equals $2.4 \cdot 10^8$ Hz. This differs by 6 orders of magnitude. This means that when the same resolution is required by a radio telescope, that an optical telescope can provide, the diameter of the radio telescope has to be 6 orders of magnitude larger. To put this into perspective, given an

optical telescope with a diameter of 10 cm, this would require a radio telescope with a diameter of 100 km to achieve the same resolution.

Because of this, radio telescopes are usually made up of many small telescopes that are placed apart and then combined into one large telescope. This way, they can achieve the required resolution. Combining many small telescopes into one large telescope is computationally heavy, so modern hardware is required for this process.

2.2 LOFAR

LOFAR (LOw-Frequency ARray) is a distributed software radio telescope network designed to detect frequencies from 30 to 240 MHz. It is primarily located in the Netherlands, but includes stations spread across Europe. It was the largest distributed radio telescope in the world before the SKA (Square Kilometre Array) and served as a test bed for it [4]. LOFAR is a distributed telescope. This means that the telescope is divided into stations that are spread apart to increase the resolution and mitigate noise. LOFAR does a considerable amount of processing at each station, but most of the work is done centrally at Groningen, the Netherlands.

Figure 1: Superterp LOFAR station - This is the central LOFAR station. Source: WikiPedia

2.2.1 LOFAR stations

Each LOFAR station contains multiple, relatively inexpensive antennas, unlike more traditional telescopes. These antennas do not move toward part of the sky. This is all done in software, allowing for fast focus and also for focusing on different parts of the sky simultaneously. There are two types of antennas, low-band antennas (LBAs) and high-band antennas (HBAs). The HBAs are the squares in Figure 1. Each LBA is a pole in the ground, where four wires reach from the top of the pole, to four corners on the ground. The LBAs detect radio waves from 10-90 MHz. This is cropped to 30-80 MHz due to interference at the lower and higher frequencies. The HBA can detect 110-250 MHz, this is again limited to 110-240 MHz due to interference. As can be seen, the spectrum of

80-110 MHz is completely missing. This is because the FM radio operates in this range, making these frequencies too polluted to be used. Both the HBAs and LBAs have two polarizations.

Each station also has a shielded cabinet that houses electronics to process the analogue signals from the antennas and turn them into digital signals that are sent over fibre optical cables to Groningen for further processing.

Each antenna is connected with a coaxial cable to the so-called receiver unit (RCU). This receiver unit reads out the analogue value into a 14 bit digital value. It does so at a sampling rate of 160 or 200 MHz. The digitized signal is then sent to the remote station processing unit (RSP). This unit turns the values into 512 complex-valued sub-bands with a polyphase filter bank, currently implemented on an FPGA.

2.2.2 LOFAR Pipeline

Figure 2 describes the central processing pipeline located in Groningen. This pipeline is all executed in real time on the incoming data, without storage to disk. This is because of the amount of data that has to be processed, which, in its unprocessed form, is not feasible to store on disk. After the real time processing pipeline, the amount of data is reduced enough to store on disk.

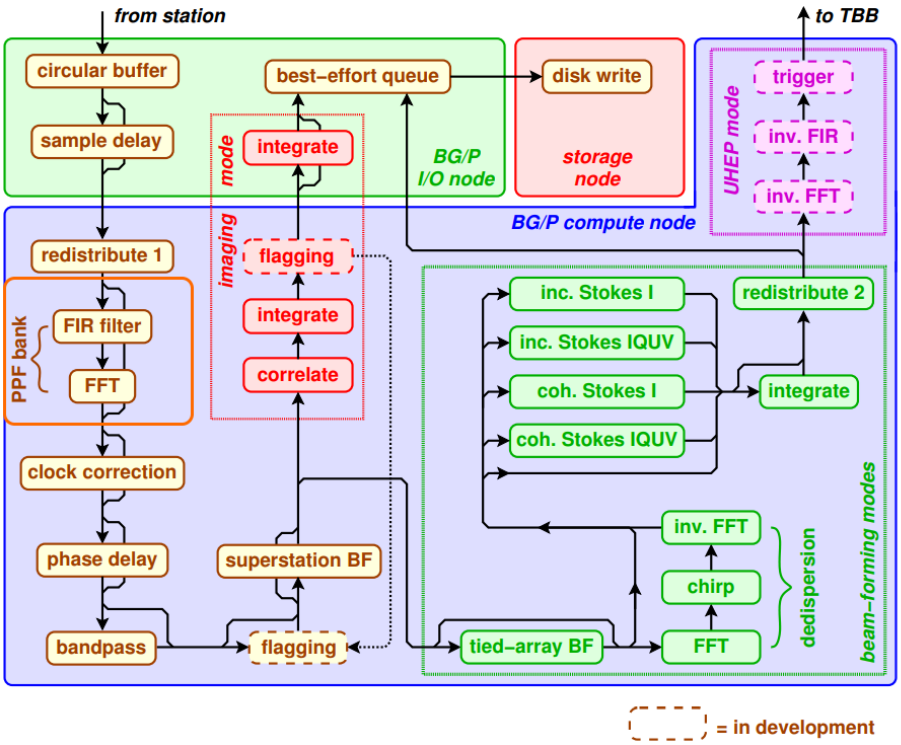

Figure 2: The LOFAR pipeline. The part circled in orange is the polyphase filter, which this thesis focuses on. Source: [4]

One of the first steps that happens centrally is the PPF bank, circled with orange. In this stage, the incoming sub-bands are split into 256 channels. The sub-bands have a width of 156 or 195 kHz depending on the sample rate selected. Each outgoing channel has a width of 0.609 or 0.761 kHz.

2.3 Polyphase Filter Bank

A polyphase filter bank (PFB) can be seen as a prism implemented in software, as visualized in Figure 3. A PFB turns multiple samples from one wider band into multiple smaller sub-bands. It consists of two stages. First, a polyphase finite impulse response filter is applied, followed by a Fourier transform.

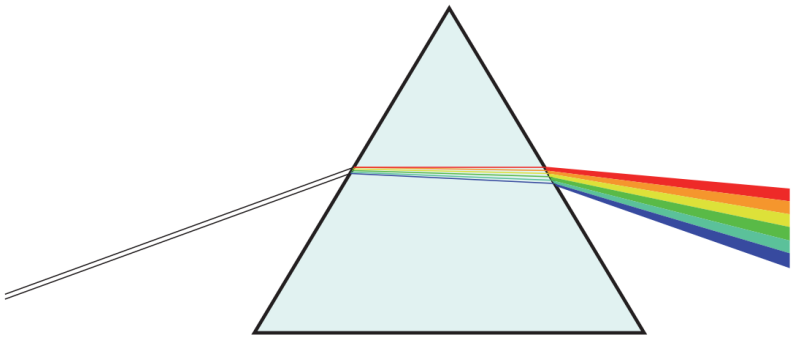

Figure 3: A prism with light passing through. A polyphase filter can be seen as a prism in software Source: [5]

2.3.1 Polyphase FIR Filter

The first step of a polyphase finite impulse response filter [6] is to downsample the incoming signal into subsequences. Each subsequences is then fed into a finite impulse response (FIR) filter. The downsampling works by dividing the incoming sequence into P subsequences. As an example, let $P = 2$ and $x(n)$ be the incoming signal. The samples are then divided among the P subsequences as follows:

$$\begin{aligned} x_0 &= \{x(0), x(2), \dots\} \\ x_1 &= \{x(1), x(3), \dots\} \end{aligned} \tag{2.4}$$

Each subsequence has its own unique FIR filter with unique weights. These separate FIR filters combined are called a polyphase FIR filter. A FIR filter is a weighted average over historical samples. The amount of samples used are called taps. A single FIR filter can be represented with the following equation:

$$y_p(n) = \sum_{m=0}^{M-1} h_p(k) x_p(n-m) \tag{2.5}$$

Here, y_p is the output of one FIR filter. There are P different outputs. x_p is the (historical) array of samples for a subsequence and h the weights for the FIR filter. The weights h for the FIR filter

are computed using a windowing function. All the outputs $y_p, \forall p \in \{0, \ldots, P-1\}$ are then fed into a P point FFT or DFT.

2.3.2 Discrete Fourier Transform

When describing a discrete Fourier transform (DFT), it is important to first understand the Fourier transform and what it does.

A Fourier transform (FT) is a method that transforms a function from the time domain into a function in the frequency domain, describing how strongly each frequency is represented in the original time domain. The general formula is given by equation (2.6) with $f(t)$ as the original function and ω the angular frequency. The function F is a complex-valued function.

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt \tag{2.6}$$

A Fourier transform is used when there is a continuous function that has to be transformed to the frequency domain. In the case of radio waves, we have only access to discrete samples. This is where the discrete Fourier transform is used.

The discrete Fourier transform (DFT) works with discrete values instead of functions. It transforms values from the time domain into complex values in the frequency domain. This is done with equation (2.7).

$$\begin{aligned} X_k &= \sum_{p=0}^{P-1} x(p) \cdot W^{kp} && \text{with } k \in \{0, 1, \ldots, N-1\} \\ W &= e^{2\pi i / P} && (W \text{ the twiddle factor}) \end{aligned} \tag{2.7}$$

In the DFT equation, X_k is the k-th complex-valued output and $x(p)$ is the p-th input, with P the number of points to transform. As can be seen, each output X_k is computed as a sum over all inputs $x(p)$ multiplied by the twiddle factor. Because p, k are zero indexed, there are $P \cdot P$ multiplications being performed, giving a complexity of $P^2 \in O(P^2)$.

2.3.3 Fast Fourier Transform

Since a complexity of $O(P^2)$ is not practical for a larger P, we also use a widely used optimized version of the DFT called the fast Fourier transform (FFT). One of the most well-known versions is an algorithm described by Cooley and Tukey [7]. Cooley-Tukey works by splitting a P point FFT into two smaller FFTs with r_1 and r_2 points, where $P = r_1 \cdot r_2$ and r is called the radix. The two smaller FFTs can then be combined to obtain the result of the larger FFT. The splitting of the FFT can be applied recursively until an FFT of size 1 is reached, which just equals the input value. Cooley-Tukey generalizes this input for any r_1 and r_2 where $P = r_1 \cdot r_2$ still holds. With a constant

radix r, Cooley-Tukey achieves a complexity of $rP \log_r P = rP\frac{\lg P}{\lg r} = \frac{r}{\lg r} P \lg P \in O(P \lg P)$. One limitation of using an FFT is that the samples have to be regularly spaced in time. With a DFT, irregular sampled data can still be transformed to the frequency domain when another value for the twiddle factor is used. While the optimizations of an FFT assume regular spacing, to effectively reuse the twiddle factors.

We choose to work only with $P = 2^n$. This greatly simplifies the algorithm. In *Numerical Recipes 3rd Edition: The Art of Scientific Computing* [8, p. 609] a very intuitive explanation is provided using the Danielson-Lanczos lemma [9]. This is done by splitting the FFT of length P into two smaller FFTs with length $P/2$. The explanation goes as follows, starting with equation (2.7) and rewriting it.

$$
\begin{aligned}
X_k &= \sum_{p=0}^{P-1} x(p) \cdot e^{2kp\pi i/P} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{2k(2p)\pi i/P} + \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2k(2p+1)\pi i/P} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{2kp\pi i/(P/2)} + W^k \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2kp\pi i/(P/2)} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{2kp(2\pi i/P)} + W^k \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2kp(2\pi i/P)} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot W^{2kp} + W^k \sum_{p=0}^{P/2-1} x(2p+1) \cdot W^{2kp} \\
&= X_k^e + W^k X_k^o
\end{aligned}
\tag{2.8}
$$

Where X^e is the FFT of the evenly indexed values in x and X^o are the values with odd indexes. Now, this derivation is also performed for $X_{k+P/2}$.

$$
\begin{aligned}
X_{k+P/2} &= \sum_{p=0}^{P-1} x(p) \cdot e^{2(k+P/2)p\pi i/P} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{2(k+P/2)(2p)\pi i/P} + \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2(k+P/2)(2p+1)\pi i/P} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{(2k2p\pi i/P)+(2P/2(2p)\pi i/P)} + \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2(k2p+k+Pp+P/2)\pi i/P} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{2k2p\pi i/P} \cdot e^{2P/2(2p)\pi i/P} + \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2k2p\pi i/P} \cdot e^{2k\pi i/P} \cdot e^{2p\pi i} \cdot e^{\pi i} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot e^{2k2p\pi i/P} \cdot e^{2p\pi i} + e^{2k\pi i/P} \cdot \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2k2p\pi i/P} \cdot e^{2p\pi i} \cdot e^{\pi i} \\
&= \sum_{p=0}^{P/2-1} x(2p) \cdot W^{2k} \cdot 1 + W^k \cdot \sum_{p=0}^{P/2-1} x(2p+1) \cdot e^{2k2p\pi i/P} \cdot 1 \cdot -1 \\
&= X_k^e - W^k X_k^o
\end{aligned}
\tag{2.9}
$$

As can be seen, X_k and $X_{k+P/2}$ have the property of using the same twiddle factor multiplication. But instead of an addition, it uses a subtraction to get to the final result. This is called a butterfly operation.

Like with the method given by Cooley and Tukey, Danielson-Lanczos can be applied recursively. So X^e can be split again, dividing it into X^{ee} and X^{eo}. This can be applied repeatedly until a one-point FFT is reached, which equals the input value $x(p)$.

For any X_k, after splitting it recursively between odd and even parts, the p turns out to be the bit-reverse value of k. This is why most FFT programs first reorder the input into its bit-reverse order. This is called a decimation in time (DIT) FFT.

2.4 Neural Networks

Neural networks are inspired by the way the human brain works. It utilizes concepts such as neurons and uses a mathematical simplification of how they work. This relatively simple idea has been expanded upon and turned into complex structures that can model many things. We describe a few neural networks that we, with some creativity, can map the polyphase filter onto.

2.4.1 Fully Connected Layer

One of the first elements used for neural networks were fully connected (FC) layers [10]. This is also one of the most simple layers, it connects every input neuron x to every output neuron s by a weight. This can be expressed with the following equation:

$$s_j = \sum_{i}^{I-1} (w_{ij} x_i) + b_j \tag{2.10}$$

With s_j the j-th output for all $j \in \{0, 1, \ldots, J-1\}$, x_i the i-th input for all inputs $i \in \{0, 1, \ldots, J-1\}$, w the weight matrix and b_j a bias addition. The weight matrix w must be in $\mathbb{R}^{I \times J}$. This equation is equivalent to a matrix ($\mathbf{W}$) vector ($\mathbf{x}$) multiplication with an addition of a bias vector $\mathbf{b}$.

2.4.2 Convolutional Neural Network

Convolutional neural networks (CNNs) are similar to FC layers in that they perform matrix multiplications of some form. The difference, however, is the amount of unique weights they use. CNNs use a so called kernel that moves along the input data to generate the output values. A kernel is a fixed weight matrix, that gets applied to one (or multiple) input sample at a time to generate one output sample. This network was first shown to be very successful in a paper by LeCun, *et al* [11] and is now widely used.

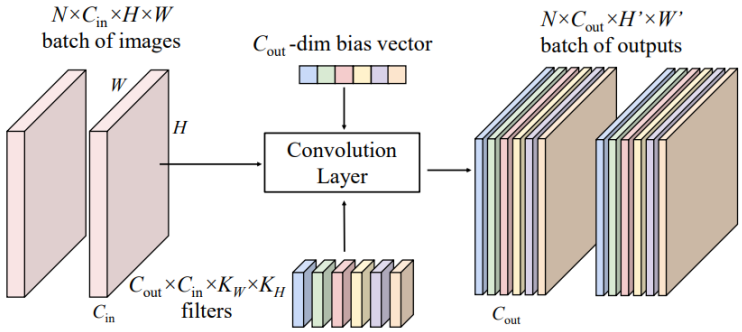

Figure 4: A visual representation of a convolutional neural network (CNN). It shows a batch of images with C_{in} channels being transformed into C_{out} channels. The weight matrices that the channels are multiplied by are shown in colour, in addition to the biases used. Source: [10]

When dealing with image data, for which a CNN was first used, the dimensions of the input and output tensors have standardized names. The first dimension is called the channel dimension C, followed by the width W and height H dimensions, as can be seen in Figure 4. Mathematically, a CNN can be expressed as follows [10, p. 292]:

$$s(i, j, c_2) = \sum_{c_1 \in \{C_{in}\}} \sum_{k,l \in \mathcal{N}} w(k, l, c_1, c_2) x(i + k, j + l, c_1) + b(c_2) \tag{2.11}$$

Here, x denotes the input data, w the weight tensor, C_{in} are the input channels and c_2 is an output channel in C_{out}. The values k, l are the offsets in the kernel used. The offsets possible for a specific kernel are contained in $\mathcal{N}$.

Note that for each output channel c_2, the outer sum goes over all the input channels c_1. In this way, a CNN combines all the input channels. If the combination of the input channels is not desired, the input channels can be split into groups. This is achieved by modifying the outer summation to only sum over a subset of channels, instead of the full set. If the number of groups equals the number of input channels, then it is called depthwise convolution.

When changing the kernel size to 1×1, the kernel is reduced to a single point. As a result, the inner loop can be omitted, because k and l are constants in this case. This way of using a CNN is called pointwise convolution. It is used for channel-wise operations, keeping the W and H dimensions separate.

2.4.3 PyTorch

One of the most popular ways to build such neural networks is by using PyTorch [12]. PyTorch is a Python deep learning framework. It allows for easy development, while not compromising on performance.

It works mainly by performing operations on tensors, which are large multidimensional arrays. These operations are executed by highly optimized C code, or they are offloaded to a GPU.

PyTorch supports layers like CNN, fully connected layers, activation layers such as ReLu, normalization layers and much more. Moreover, it also supports creating custom layers with matrix operations. These layers can then be combined in a so-called model.

A model can then be trained using backpropagation to learn its weights. The weights of a model can also be set directly. This opens up ways to use PyTorch layers for operations different from what they were meant for. For example, if the bias in a CNN is set to zero, what is left is one giant multiplication and addition by the weights of the kernel. The same can be done with a fully connected layer. If the bias is set to zero, what is left is a matrix vector multiplication.

An important part of PyTorch is the way it standardizes its use of dimensions. The first dimension is called the batch dimension. Using the batch dimension (N) generally increases performance because the same neural network layer is applied to a whole batch at the same time, allowing for data parallelism. The next dimensions have less standardized names because they depend on what kind of data is used. When using image data, the next dimensions are usually called the channel (C), height (H) and width (W) dimension.

2.4.4 ONNX

ONNX (Open Neural Network Exchange) [13] is a format for representing (trained) machine learning models. This format allows for transfer between different frameworks. Frameworks like PyTorch, TensorFlow and TensorRT (see Section 2.5) support exporting or importing ONNX models, allowing models to be created in one framework and imported in another.

ONNX has different versions. As of writing this, ONNX is on version 1.19, also called version 19. The different versions have different ways of representing specific operations/nodes. The ONNX specification states that every node in a graph must have the same version, making it important to choose a version that suits all requirements while still being supported.

There are many different ONNX operators, or more generally called, layers. We describe some that we deem useful for multiplications and additions of a PFB. After that, a few layers are listed that can be used to reorder the data in the tensors. Reordering of the data is important because all layers have strict requirements of the input data layout.

- `Conv` - The Conv operator is the CNN as described in section 2.4.2. It supports setting the number of groups, as well as omitting the bias addition, making it fully compatible with the CNN equation.
- `Add` - This operator allows for the addition of tensors $\mathbf{A}$ to $\mathbf{B}$. ONNX also has an operator for the subtraction of tensor $\mathbf{B}$ from $\mathbf{A}$. This is the `Sub` layer. Add and Sub also supports broadcasting. Broadcasting can be done when tensor $\mathbf{A}$ has one less dimension than $\mathbf{B}$, but the rest of the dimensions match. It then copies tensor $\mathbf{B}$ over that missing dimension, ensuring the dimensions match again.
- `MatMul` - The MatMul layer performs a matrix multiplication between the input matrix $\mathbf{A}$ and $\mathbf{B}$. A fully connected layer, as described in section 2.4.1 decomposes into a MatMul layer, followed by an Add layer, when bias addition is enabled.
- `MatMul` - This layer is an element-wise operation similar to the Add and Sub layers. Multiply each value in the input matrix $\mathbf{A}$ by the value with the same index in the matrix $\mathbf{B}$. It also supports broadcasting.
- `Gemm` - Gemm performs a matrix multiplication plus addition. Given input matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, it computes $\mathbf{Y} = \mathbf{AB} + \mathbf{C}$. Fully connected layers can also be transformed into Gemm instead of MatMul and Add.
- `AveragePool` - The AveragePool layer can takes the average of a window, which walks along the input. The window functions in much the same way as a kernel from a CNN. Average pool is used to average values and reduce the size of a dimension.
- `Gather` - Gather allows for the selection of elements from a layer along an axis. As input, it requires a tensor $\mathbf{T}$ and a list of indices $\mathbf{i}$. This selection remains true to the order in which the elements are specified in the $\mathbf{i}$ array. If all input indices are selected, the output is a reordered version of the input.
- `Split` - The Split operator enables the splitting of a tensor along a specified axis. It also allows for setting the length of each output. Let us look at an example for a tensor with dimensions (N, C, H, W) where $C = 10$. When this tensor is split along axis C, this results in 10 tensors of shape $(N, 1, H, W)$.
- `Concat` - The Concat operator enables the concatenation of tensors along an axis. The input is a list of tensors to concatenate, the output is one tensor that contains the concatenated input.
- `Transpose` The Transpose layer allows for transposing a matrix ($T_{out} = T_{in}^T$). It allows for an optional argument to select which channel to transpose along, if the input is an tensor instead of a matrix.

- `Reshape` - This operator enables modification of the shape of the input tensor. Conceptually, it flattens the tensor into an array and then fills the new shape specified in row-major order.
- `Constant` - This layer contains constant values. This is used, for example, when a fully connected layer is transformed into ONNX. A fully connected layer can be represented as a MatMul and an Add layer. The second input values from MatMul and Add are both constant in that case, so are stored in a Constant layer. Weights for the Conv layers are stored in the convolution layer itself.

2.5 TensorRT

TensorRT, as stated by NVIDIA, is an ecosystem of tools to achieve high-performance deep learning inferences [14]. It contains tools to accelerate machine learning applications, as well as tools to perform inference on NVDLA with. In this thesis, the `trtexec` tool is used. This tool can be used to perform inferences of ONNX models on NVDLA. It also allows for the compilation of ONNX models to a format that is compatible with cuDLA, which will be explained further in later sections.

2.6 CUDA

For general purpose applications that benefit from GPU acceleration, NVIDIA has CUDA [15]. CUDA enables the development of applications that can be executed on NVIDIA GPUs. With CUDA, so-called kernels can be written, which are small C programs that can be executed on the GPU. In order to make the use, and synchronization, of kernels easier, CUDA introduces streams.

A CUDA stream can be thought of as a FIFO queue. Each object that is enqueued on the stream is executed in FIFO order. Operations such as `cudaMemcpyAsync`, which copies data between the host and device (GPU), can also be submitted on the stream. When every operation is submitted on the same stream, CUDA ensures that execution occurs in the correct order, without the need for synchronization on the developers side. Streams also enable async executions, allowing the CPU to do other work.

2.7 NVDLA

NVDLA (NVIDIA Deep Learning Accelerator) [16] is an architecture to accelerate machine learning inference. It is created by NVIDIA and subsequently released as open source. It consists of a software stack, a hardware specification and a Verilog model. NVIDIA is currently using it on the NVIDIA Jetson line.

2.7.1 Hardware

There are two main hardware versions of NVDLA. Version 1 is a non-configurable version of NVDLA. Version 2 is a scalable design that has the same subunit designs as NVDLA v1. The following description is of the NVDLA v1 standard. NVDLA v2 also supports a hardware scheduler such as a microcontroller, while on NVDLA 1, all scheduling is done by the CPU.

NVDLA consists of multiple subunits. The main subunits are the bridge DMA, convolution pipeline, single data point processor, planar data processor, cross channel data processor, RUBIK module,

MCIF and SRAMIF modules. NVDLA operates on INT8 or FP16 data types.

Bridge DMA In order to use the internal SRAM that an NVDLA contains, NVDLA requires a subunit that moves data between internal SRAM and external DRAM and the other way around. This is where the bridge DMA (BDMA) comes in. The BDMA contains two DMA interfaces, both connected to the SRAM and DRAM. Both interfaces have a width of 512 bits.

Convolution Pipeline In NVDLA, the convolution pipeline is the most advanced pipeline. This is understandable, as most of the processing is spent on CNNs for traditional machine learning applications such as image classification or detection. The pipeline consists of several stages, as can be seen in Figure 5.

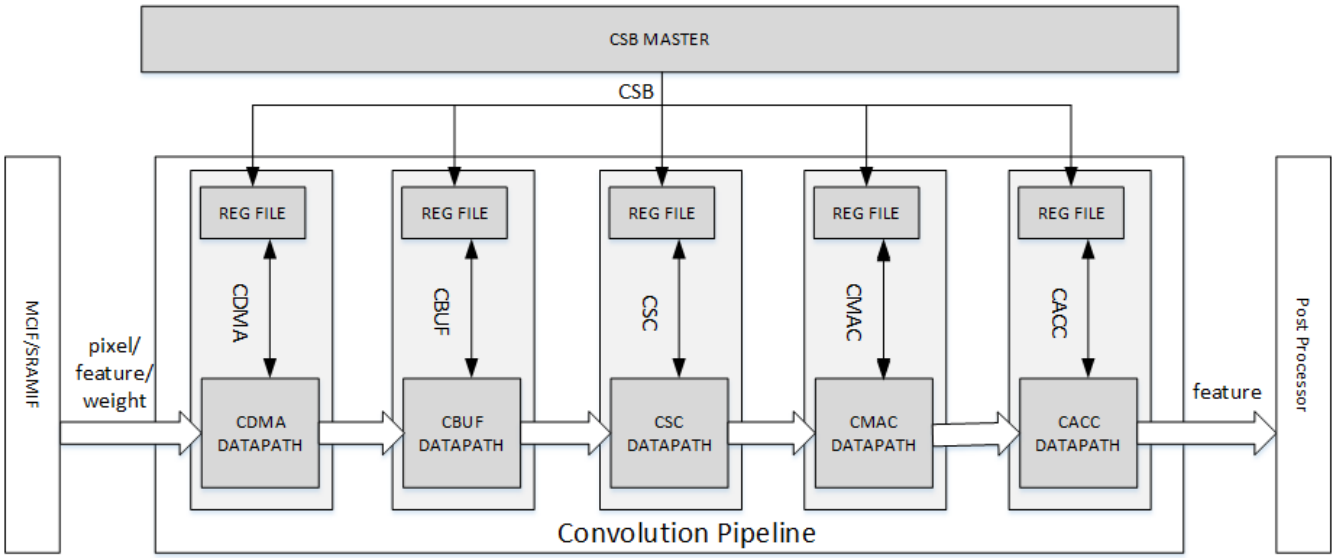

Figure 5: NVDLA convolution pipeline. Source: https://nvdla.org/.

As stated above, the Configure Space Bus (CSB) master is the CPU in NVDLA v1, and a hardware unit in v2. This unit schedules and configures each stage in the convolution pipeline, but the CPU remains in control of some background tasks.

The convolution pipeline has three main modes of operation. Direct convolution, image convolution and Winograd convolution. The last two are special cases of direct convolution. Winograd convolution is an optimization for direct convolution, which decreases the number of multiplications required. NVDLA supports Winograd for kernels of size 3x3xC, where C stand for channel. Image convolution is used if the input data type is an image. It supports pixel values, while direct convolution and Winograd do not support this as an input data type. NVDLA supports sparse weight compression for direct convolution to reduce memory bandwidth.

The first stage in the convolution pipeline is convolution direct memory access (CDMA). This stage fetches data from DRAM or SRAM and moves it to the convolution buffer. The CMDA supports moving feature data, pixel data, weight data, weight mask bit group (WMB) and more into the convolution buffer.

The second stage is the convolution buffer (CBUF). It is a 512KB SRAM device. It can be used as a cache for input data and weight data. It consists of 16 32KB banks. The CBUF works as a circulair buffer. New data is put at increasing indexes. If the weight data is uncompressed, all 16 banks are used for the weight and input data. If the data is compressed with a weight mask bit group (WMB), the WMB is stored in bank 15 and the rest can be used for input data or weight data.

The third stage is the convolution sequence controller (CSC). This stage selects input and weight data from the CBUF and sends it to the convolution MAC unit. It also schedules the CMAC.

The fourth stage of the convolution pipeline is the convolution MAC (CMAC). This stage receives its data from the CSC. The CSC contains identical submodules that perform the actual multiplications. The amount of submodules and how many multipliers each subunit contains depend on the implementation of NVDLA v2. The output of the CMAC gets send to the accumulator.

The fifth and last stage is the convolution accumulator (CACC). The CMAC writes its output to the CACC where the CACC accumulates these partial sums. The CACC has a large buffer, that can also function to smooth out peaks in throughput in the pipeline.

Single Data Point Processor The single-point data processor (SDP) is mostly used for post-processing after the convolution pipeline. It handles bias addition, non-linear functions such as activation functions and batch normalization. It also manages the element-wise operations like addition, subtraction or multiplication.

Planar Data Processor The planar data processor (PDP) is used to execute operations that work on planes of data. It performs operations such as min or max pooling layers. It works on the height $\times$ width dimensions of the data.

Cross Channel Data Processor Since the PDP works along the height and width dimensions, the PDP cannot normalize along the channel direction. This is where the cross-channel data processor (CDP) comes in. It enables the use of layers such as local response normalization (LRN).

RUBIK module The RUBIK module works much like the BDMA. It never modifies the data, it just transfers it. There are three different operations that the RUBIK module can perform. The first operation is contract. The contract operation is used by deconvolution, which will be skipped here.

The second and third operations are inverse operations of each other. They are split and merge. Split splits a data cube that is in the format (C, H, W) along the C axis into planes of shape (H, W). Merge does the reverse; it merges a list of C planes with shape (H, W) into a cube of shape (C, H, W).

MCIF and SRAMIF modules The memory controller interface (MCIF) and SRAM interface (SRAMIF) connect the submodules to an AXI bus, which provides connection to the external DRAM or internal SRAM.

2.7.2 Software

At a higher level, the NVDLA software stack consists of two parts. There is an online and an offline part. In the offline part, the model is defined and compiled. In the online part, inference happens. A global overview is given in Figure 6. The online part can be seen on the right of the dashed line. The User Mode Driver and the Kernel Mode Driver have both been made open source. TensorRT is partly open source, but the most critical parts are closed source.

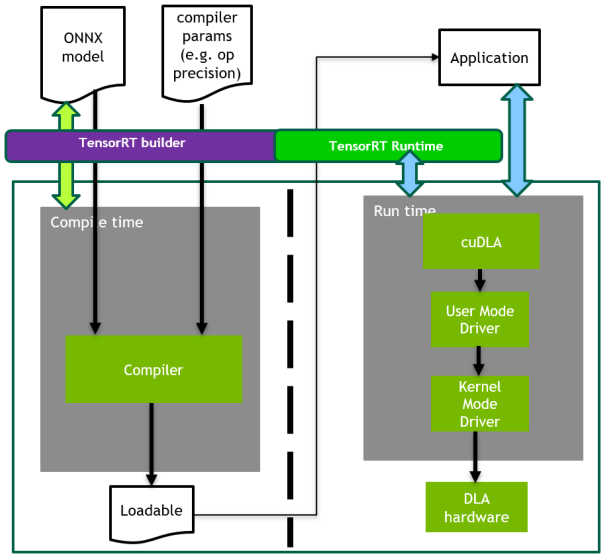

Figure 6: The NVDLA software stack. The part on the left of the dashed line is the offline part, the part on the right is online. Source: [14]

Compiler The compiler is the last step in the offline process. There are currently two compilers from NVIDIA for NVDLA. A compiler in the TensorRT ecosystem (`trtexec`) and an old Caffe [17] compiler. Since the Caffe compiler is no longer maintained and Caffe has been merged with PyTorch, the TensorRT compiler is the logical option. The TensorRT compiler takes an ONNX model and can generate a NVDLA loadable. A small note on this is, that the Caffe compiler has been made open source. In theory, one could make their own compiler. However, the code is undocumented and thus not easy to follow.

Loadable Although the compiler for the Caffe models is now open source, there is still no specification for the NVDLA loadable file format anywhere to be found. There is a FlatBuffers schema file in the open source GitHub repository that describes the loadable [2], this is, however, only a very global representation and does not contain the objects required to directly write your own loadables. What is described in the documentation is that, like the ONNX representation, a loadable can be seen as a dependency graph. Each node in the graph is executed on a pipeline or subunits. In the repository accompanying this thesis there is a subfolder `lb_reveng`, that contains some Python files that attempt to extract useful information about a given loadable.

cuDLA cuDLA [18] is an extension of CUDA for NVDLA. cuDLA allows the execution of NVDLA loadables in a CUDA environment. When executing a loadable, the following steps have to be done. First, the cuDLA device has to be initialized. This is done with the `cudlaCreateDevice` function. After this is successful, a `cudlaDevHandle` is created. This devHandle can then be used to load a NVDLA loadable from memory with `cudlaModuleLoadFromMemory`. Once the module is loaded, there are several functions to get information about the module, mainly about the input

[2] https://github.com/nvdla/sw/blob/master/umd/core/src/common/loadable.fbs

and output data format and dimensions. A devHandle only supports the loading of one loadable and one physical device supports 16 concurrent device handles.

Next, memory must be allocated with the CUDA API using `cudaMalloc`. Afterwards, the memory has to be registered with the IOMMU for DLA use, this can be done with `cudlaMemRegister`. Data can then be transferred with any CUDA function for copying data from the host to the device or back.

With memory and a model, a `cudlaTask` can be created. The cudlaTask struct contains a pointer to the model used, pointers to the input and output memory and optionally, events used synchronisation. A task can then be executed with `cudlaSubmitTask`. This submits a task to a CUDA stream, which allows for easy async execution and interleaving with CUDA kernels.

User Mode Driver The User Mode Driver (UMD) is responsible for providing an API to load loadables and registering memory locations. cuDLA is a thin wrapper around this functionality. The UMD then passes the neural network to the KMD for execution.

Kernel Mode Driver The Kernel Mode Driver (KMD) is then responsible for execution of the module. It parses the dependency graph and decides how to schedule the layers. The KMD also communicates with the scheduler, which handles fine-grained control over the execution. The scheduler then interrupts the KMD when it is done with its work.

Layers and limitations TensorRT itself does not support all ONNX layers. Furthermore, the layers it supports on NVDLA are a subset of the supported layers on TensorRT. The layers supported on TensorRT also have restrictions. One general limitation of NVDLA is that it requires all data to be in the shape of (N, C, H, W), with N the batch dimension, C the channel dimension and H, W the height and width. A second limitation is that the batch dimension N is limited to a size of 4096 and each other dimension has a maximum size of 8192.

Below is a list of the layers described in Section 2.4.4 and an explanation of their limitations when used with TensorRT on NVDLA. The current status of which layers are supported can be found in the *NVIDIA Deep-Leraning-Acceleraotr-SW* repository on GitHub [3]. The limitations for these supported layers can then be found in the TensorRT documentation [14].

- Conv - Convolution is supported extensively, including setting groups. Two limiting factors that must be taken into account are the maximum kernel size and the maximum supported groups. The maximum kernel size cannot exceed 32. The group size must not exceed the maximum supported channels, which equals 8192.
- Add & Mul - These are both so-called ElementWise layers. If broadcasting is required, it has some limitations on the input shapes. Broadcasting works if you multiply one matrix by a second matrix that is missing the first dimension, but has the same shape for the other dimensions. It can then broadcast the smaller matrix over the mission dimension. When broadcasting is required, the dimensions must be in the form (N, C, H, W), which means that they are all of equal length. $(N, C, 1, 1)$ or $(N, 1, 1, 1)$

[3] https://github.com/NVIDIA/Deep-Learning-Accelerator-SW/blob/main/operators/README.md

- MatMul - Not supported, can be reconstructed with pointwise convolution.
- Gemm - Not supported, requires input from Constant layer, which is not supported
- AveragePool - Supported, but limited to a window of size 8.
- Gather - Can be reconstructed. This reconstruction is done with pointwise convolution. This means that a Gather layer of n points turns into a n^2 operation.
- Split & Concat - Both supported, but not along the batch (N) axis.
- Transpose & Reshape - Supported, but again not along the batch (N) dimension.
- Constant - Not supported.

2.8 PowerSensor3

The PowerSensor3 [19] is a power sensor developed for measuring the power draw of SoC devices and PCIe cards. It can sample at a rate up to 20 kHz. With such a high sampling rate, it can give insights into the power consumption of very short-lived events, such as the execution of a CUDA kernel.

Theoretically, the PowerSensor3 has an error of ±7.0 W via USB-C, which is quite high. However, in section *B. Long term stability* long term testing was done, which showed that averaging over multiple samples gave fluctuations of ±0.09 W (at 128 k samples).

There are multiple ways to read the data from the PowerSensor3. There are a few simple CLI applications and a C++ API. There are also Python bindings. The source code can be found on the GitHub repository [4] of the PowerSensor3.

The API has two main ways of being used. The first way is to dump all the samples into a file and the second is to measure energy consumption between multiple points in time. Writing all of the samples to a file can be done at a fixed sampling rate. The API also allows for the insertion of markers into the dump file. The insertion of markers allows for the analysis of power draw at certain points in the execution of an application. The second way of using the API, is useful for determining energy consumption between points in time. With energy consumption, one can determine the energy efficiency of an implementation.

3 Related work

There has been research into using deep learning accelerators for other applications than neural networks. However, most deep learning accelerators differ in the subset of neural network operations they support, making portability difficult. More general research into the use of accelerators, such as GPUs, for PFB specifically is also available.

3.1 TINA

[4] https://github.com/nlesc-recruit/PowerSensor3

TINA [20] is a framework for implementing signal processing algorithms on neural network accelerators. The paper investigates how to use methods such as depthwise convolution, pointwise convolution and standard convolution to perform signal processing tasks that perform faster than CPU implementations.

For developing TINA, PyTorch is also used. There is a GitHub repository [5] that contains examples of the TINA layers. Some of the functions that are implemented in TINA are matrix-matrix multiplication, element-wise matrix addition and summation, a DFT, an inverse DFT and an FIR filter.

The DFT and FIR filter are of particular interest, as this matches the aim of this thesis. The approach used in TINA for the implementation of the FIR filter is the same as used in this thesis. However, in their paper, they use normal convolution for a FIR filter, while in this implementation, depthwise convolution is used for a whole polyphase FIR filter.

However, the TINA DFT is a real-to-complex DFT. As we require a complex-to-complex DFT for our application, the DFT is not transferable.

3.2 Coral FFT

Another project of interest is the implementation of an complex-to-complex FFT on the Coral TPU [21]. This project is written in TensorFlow and can also be found on GitHub [6]. With slight modifications, this model can also be exported to ONNX, as visualized in Figure 7.

In this visualization, a 8 point FFT is shown. The dimensions are (N, P, Z) with N the batch dimension, P the points and Z the real and complex parts. As can also be seen, this model makes heavy use of squeeze, split and concatenate to re-order the data. It does this along axis other than the channel axis (index 1, so the P axis here), this is not supported in TensorRT. It also uses squeeze, which is not mentioned in section 2.7.2, but is not supported on NVDLA. The way it uses the different dimensions also does not match the requirements of NVDLA. All of this makes this model unable to be ported to work on NVDLA.

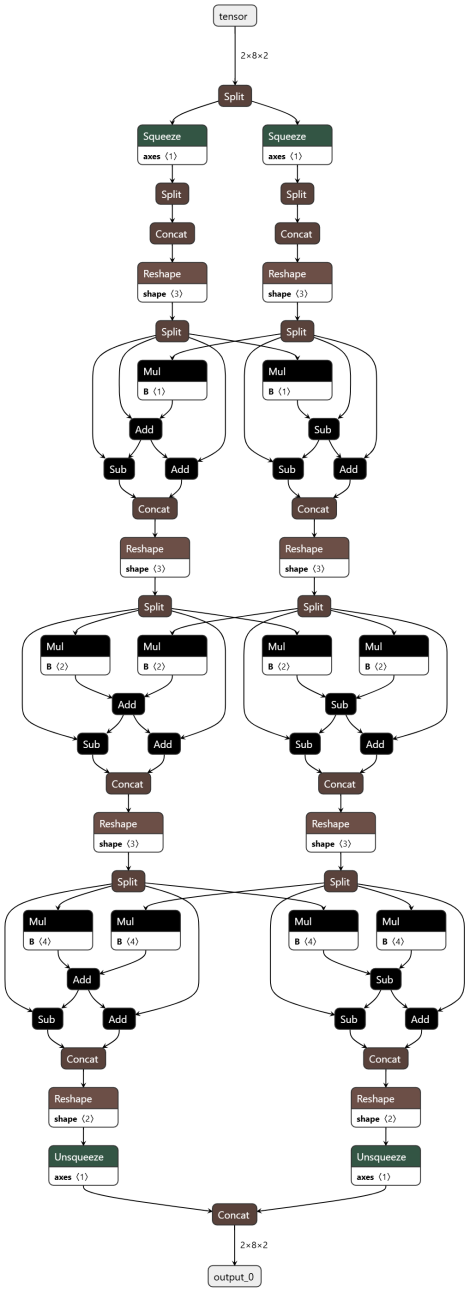

Figure 7: A visualization of the Coral FFT. This model performs a 8 point complex-to-complex FFT with batch size 2. The real and complex values are separated in the last dimension. Image generated with Netron.

[5] https://github.com/ChristiaanBoe/TINA
[6] https://github.com/OliVis/coral/tree/main

3.3 LOFAR GPU Kernel

In 2012 [1], a paper investigating the effectiveness of a GPU implementation of a polyphase filter bank was published. The current LOFAR pipeline relies heavily on GPU acceleration. The new design, however, has not been published.

In order to compare the polyphase filter on NVDLA to the current GPU implementation, some quick benchmarks were performed on a Jetson Orin AGX 64GB. This is the same model used for later testing of NVDLA. The throughput was 6.7 billion samples per second, at a data rate of 80.1 GB/s. This is for a PFB with 256 channels and 16 taps.

The next step in the LOFAR pipeline is the correlator. The Jetson can do the correlation part of the pipeline at 6.1 billion samples per second.

3.4 Reference Polyphase Filter Bank

For verification of the correctness of the implementation, the stand-alone version [22] of the LOFAR polyphase filter bank is used. This reference implementation contains code to generate weights for different kinds of filtering of windows. It also includes a polyphase FIR implementation based on an array to keep track of historical samples. The FFT is performed with fftw3 [7].

4 Design and Implementation

For each part of the polyphase filter bank, this section first describes theoretically how it can be implemented for the use on deep learning acceleration hardware, which means mapping all logic to operations on matrices. Secondly, it gives an implementation of each part that works on NVDLA. Although the design sections are generally widely applicable, the implementation section focuses on the NVDLA limitations specifically, also taking into account that we require an PFB that has 16 taps and produces 256 channels.

4.1 Implementation Decisions

One part of the implementation that is glossed over in the design phases of the following sections is the requirement for a complex-to-complex PFB. As complex numbers are generally not supported by neural networks, the equations have to be split into real and imaginary parts. Then these parts have to be computed with real numbers. This also requires a decision on how to store the complex values in memory. We choose to interleave the real and imaginary parts. This means that an array $X \in \mathbb{C}^n$ with complex values $[x_0, x_1, \ldots, x_n]$ is stored in memory as array $\tilde{X} \in \mathbb{R}^{2n}$ with real values $[Re(x_0), Im(x_0), Re(x_1), Im(x_1), \ldots, Re(x_n), Im(x_n)]$.

A second important distinction is that NVDLA only supports INT8 and FP16. The latter is chosen for this implementation to ensure better accuracy. NVDLA is also designed around, and optimized for, its convolution pipeline. When different neural network operations can be used to mathematically achieve the same end result, favour is given to a design using a CNN. This decision also has the benefit of wider applicability, as most of the deep learning accelerators support CNNs.

[7] https://www.fftw.org/

4.2 Polyphase FIR filter

Building upon section 2.3.1, which describes mathematically how a polyphase FIR filter works and its use. We continue with a design and implementation that ultimately works on NVDLA.

4.2.1 Design

When implemented on a GPU or CPU, some form of buffer is used to keep the historical values of each tap that an FIR filter requires. Since deep learning acceleration hardware limits the operations that can be performed to standard machine learning operations, we cannot implement some form of buffer internally on the hardware. Since we cannot use a buffer, we need to supply all the historical tap values every time we run the filter.

Let us create a polyphase FIR filter with P channels and M historical values (taps) per channel. When mapping this to operations that can be performed on matrices, the input matrix $\mathbf{X}$ must be in $\mathbb{C}^{P\times M}$. For the data to be in the correct order, the incoming samples have to be put into the matrix column-major. This means that for $\mathbf{X}(p,m)$, $\mathbf{X}(0,0) = x(0)$, $\mathbf{X}(1,0) = x(1)$, $\mathbf{X}(p,0) = x(p)$, $\mathbf{X}(p,m) = x(p + P \cdot m)$.

Recall equation (2.5), which is the equation for one FIR filter. When we rewrite the weights $h_p(k)$ as matrix $\mathbf{H} \in \mathbb{R}^{P\times M}$ we can do the following:

$$
\begin{aligned}
y_p(n) &= \sum_{m=0}^{M-1} h_p(m)x_p(n-m) && \text{Equation (2.5)} \\
y_p &= \sum_{m=0}^{M-1} h_p(m)\mathbf{X}(p,m) && \text{Where } y_p \text{ equals the latest} \\
& && \quad \text{output for channel } p \\
y_p &= \sum_{m=0}^{M-1} \mathbf{H}(p,m)\mathbf{X}(p,m)
\end{aligned}
\tag{4.1}
$$

Generalized for the whole polyphase FIR filter, which outputs can be described by vector $\mathbf{y} = \{y_0, y_1, \ldots, y_{P-1}\}$. The whole computation can be seen as an element-wise multiplication of matrix $\mathbf{H}$ by matrix $\mathbf{X}$, after which a summation is computed along the m axis.

4.2.2 Implementation

The first step is to ensure that the design described above can be used with complex numbers. Since the weight matrix $\mathbf{H}$ contains real values, the incoming complex samples are multiplied by a real value. Multiplying a complex value by a real value is the same as multiplying the real and imaginary parts of the complex value independently by the real value. This means that we can simply apply the weight from $\mathbf{H}$ to the real and complex parts of the input samples.

This is done by expanding the last line in equation (4.1) to work with the interleaved matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{2P \times M}$ or simply $\tilde{\mathbf{X}} \in \mathbb{R}^{Q \times M}$ with Q the number of stored values, so $Q = 2P$. For the equation to work with $\tilde{\mathbf{X}}$, a $\tilde{\mathbf{H}} \in \mathbb{R}^{Q \times M}$ is also required. Since the real and imaginary parts are interleaved in the first dimension, each row of $\mathbf{H}$ has to be duplicated to create $\tilde{\mathbf{H}}$. After $\tilde{\mathbf{H}}$ is computed, $\mathbf{H}(p) = \tilde{\mathbf{H}}(2p) = \tilde{\mathbf{H}}(2p+1)$ for all $p \in \{0, 1, \ldots, P-1\}$.

With both $\tilde{\mathbf{X}}, \tilde{\mathbf{H}} \in \mathbb{R}^{Q \times M}$, we can write:

$$\begin{aligned} y_p &= \sum_{m=0}^{M-1} \mathbf{H}(p, m)\mathbf{X}(p, m) \\ \tilde{y}_q &= \sum_{m=0}^{M-1} \tilde{\mathbf{H}}(q, m)\tilde{\mathbf{X}}(q, m) \end{aligned} \tag{4.2}$$

Here, $\tilde{y}_q$ are the interleaved results of one FIR filter, with q even the real parts, and q odd the imaginary parts. All separate FIR filters can again be combined to form one polyphase FIR filter, with as output an interleaved vector $\tilde{\mathbf{y}} = \{\tilde{y}_0, \tilde{y}_1, \ldots, \tilde{y}_{Q-1}\}$

Now that the computation required for complex values is known, this has to be mapped to supported neural network operations. The calculation is still an element-wise matrix multiplication of matrix $\tilde{\mathbf{H}}$ by matrix $\tilde{\mathbf{X}}$, followed by a summation across the dimension m.

Let us first explore the use of a Mul layer to perform element-wise multiplication. Mul is supported on NVDLA, however, it does come with a few limitations. As stated in section 2.7.2, when using broadcasting, the Mul operation has requirements for the shape of the dimensions. The shape must be equal to (N, C, H, W), $(N, C, 1, 1)$ or $(N, 1, 1, 1)$.

According to the mathematical formulation above, we have samples in $\mathbb{R}^{Q \times M}$. Where M equals the number of taps and $Q = 2P$ with P equals the number of channels. In the case of 16 taps and 256 channels, it gives the shape $(16, 512)$ or $(512, 16)$. Since N is the batch dimension and cannot be used when broadcasting is required, there is no way to match this shape to the requirements of the Mul operation without changing the layout of the data. Changing the layout of the data can be done, but this does require additional operations.

One way to still make use of a Mul layer, would be to not use broadcasting. This would however mean that the weight matrix $\tilde{\mathbf{H}}$ cannot be applied to a batch at a time. That would require that the weights be transferred to the NVDLA for every layer, increasing the bandwidth used for the weights by a factor equal to the batch size used otherwise.

After the element-wise multiplication, the summation needs to be computed. There is no direct summation option, but a summation can be turned into an average pooling layer, followed by multiplying each result by the amount of values over which the average was taken. The average pooling layer in NVDLA has a maximum window size of 8, which limits the number of taps to 8.

The method described above decomposes the polyphase FIR filter into many layers, including the reordering of data, which is inefficient, and not all requirements are met. As it turns out, a

convolution layer does exactly what is required. It performs multiplications by constants, followed by a summation. Let us take equation (2.11):

$$s(i,j,c_2) = \sum_{c_1 \in \{C_{in}\}} \sum_{k,l \in \mathcal{N}} w(k,l,c_1,c_2)x(i+k,j+l,c_1) + b(c_2) \tag{4.3}$$

The equation differs from the equation for an FIR filter, but we can rewrite it to suit our needs. As we only require one summation, we drop the outer summation. This can be done by splitting the convolution into groups. The amount of groups equal the number of separate channels c, so this convolution is called depthwise convolution. Note that this is a channel in the convolution, which is not exactly the same as a channel in the polyphase FIR filter. The bias addition is also discarded, as this is not required.

$$s(i,j,c) = \sum_{k,l \in \mathcal{N}} w(k,l,c)x(i+k,j+l,c) \tag{4.4}$$

Above, $\mathcal{N}$ represents the offsets in the kernel, so the convolution has a kernel of size $(\max(\mathcal{N}), \max(\mathcal{N}))$, which it performs a multiplication and summation on. For an FIR filter, only a $(1, \max(\mathcal{N}))$ kernel is required. This also effectively reduces the dimensions of s, w and x by one.

$$s(i,c) = \sum_{k \in \mathcal{N}} w(k,c)x(i+k,c) \tag{4.5}$$

In theoretical texts, the channel dimension c is usually denoted at the end. In practice, the layout is (N, C, H, W), so we switch these dimensions around.

$$s(c,i) = \sum_{k \in \mathcal{N}} w(c,k)x(c,i+k) \tag{4.6}$$

Now, with convolutional networks, the kernel walks along the input data. Each step it generates an output, so if the kernel size matches the input data, it only generates one output. If we ensure that $k \in \mathcal{N}$ goes up to the same value as the second dimension of x, $s(c,i)$ will equal $s(c,1)$, so we get:

$$s(c) = \sum_{k \in \mathcal{N}} w(c,k)x(c,k) \tag{4.7}$$

Next, we set the convolution channel c to q from equation (4.2). We also change the sum notation from $k \in \mathcal{N}$ to sum from $m = 0$ till $M - 1$

$$s(q) = \sum_{m=0}^{M-1} w(q,m)x(q,m) \tag{4.8}$$

Lastly, if we set $w(q,m)$ equal to $\tilde{\mathbf{H}}(q,k)$ and $x(q,m)$ equal to $\tilde{\mathbf{X}}(q,k)$, it matches equation (4.2) exactly with the output $s(q) = \tilde{y}_q$

In conclusion, this means that a polyphase FIR filter can be mapped to a CNN with kernel size $(1, M)$ and both convolutional channel size and groups equal to $Q = 2P$. Since NVDLA requires all four dimensions to be used for the input and output tensors, we require the input in the form of $\mathbb{R}^{N\times Q\times 1\times M}$. After the polyphase FIR filter is executed, it will return an output in the form of $\mathbb{R}^{N\times Q\times 1\times 1}$. It does this in $2P \cdot M$ multiplications, giving a complexity of $2PM \in O(PM)$. A visual representation is shown in Figure 8 of a polyphase FIR filter with a batch size of 5, 256 complex-valued channels and 16 taps.

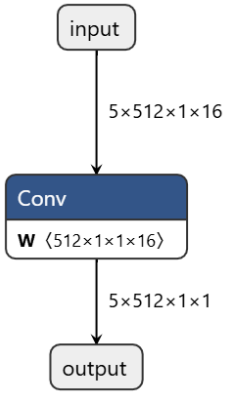

Figure 8: A 256 channel (complex-valued) polyphase FIR filter with 16 taps. It has a batch dimension of 5. Image generated with Netron.

An implementation in PyTorch of a polyphase FIR filter based on a CNN can be found in `fir_cnn_module.py` in the accompanying repository. The module supports any size of taps and channels. However, NVDLA limits the kernel size to 32, so no more than 32 taps are supported on NVDLA. The same applies to the number of groups, which is limited to the maximum dimension size of 8192. This means that the maximum number of complex-valued channels is 4096 on NVDLA.

Weight matrix The weight matrix for the polyphase FIR filter is generated by the reference implementation, which has been slightly modified for ease of use and is called from the `fir_helper.py` file.

4.3 DFT

After the polyphase FIR filter, the channels are fed into a P point complex-to-complex DFT. In this section, we propose a design for a DFT and implementation for that works on NVDLA.

4.3.1 Design

In section 2.3.2, we left off with an equation to calculate the result for one point of a DFT. Since the goal is again to map this to a neural network operations, we require rewriting it in operations performed on matrices (or vectors). An important part of the DFT equation is the twiddle factor $W = e^{2\pi i/P}$. As this twiddle factor is constant for a P-point DFT, this can be precomputed, saving computation in the inference stage. We continue with the DFT equation:

$$X_k = \sum_{p=0}^{P-1} x(p)W^{kp} \qquad \text{Equation (2.7)} \tag{4.9}$$

As the twiddle factor W is dependent on k and p, a natural way would be to write it as matrix $\mathbf{W}(k,p) \in \mathbb{C}^{P\times P}$ where $\mathbf{W}(k,p) = W^{kp} = e^{2\pi ikp/P}$.

$$X_k = \sum_{p=0}^{P-1} x(p)\mathbf{W}(k,p) \tag{4.10}$$

This can be written as a vector (x) matrix ($\mathbf{W}$) multiplication, with output vector X, but this form suits our further need.

4.3.2 Implementation

As with the polyphase FIR filter, the first step is to ensure that the design works with the representation for interleaved values that we chose. In this case $\mathbf{W}$ is a complex value, so this is a multiplication of two complex values. Let us write the multiplication of complex numbers z_v and z_c with real and imaginary parts separately.

$$\begin{aligned} z_v &= \operatorname{Re}(z_v) + \operatorname{Im}(z_v)\, i \\ z_c &= \operatorname{Re}(z_c) + \operatorname{Im}(z_c)\, i \end{aligned} \tag{4.11}$$

$$\begin{aligned} z_v z_c &= (\operatorname{Re}(z_v) + \operatorname{Im}(z_v)\, i) \cdot (\operatorname{Re}(z_c) + \operatorname{Im}(z_c)\, i) \\ &= \operatorname{Re}(z_v)\operatorname{Re}(z_c) + \operatorname{Re}(z_v)\operatorname{Im}(z_c)\, i + \operatorname{Im}(z_v)\operatorname{Re}(z_c)\, i + \operatorname{Im}(z_v)\operatorname{Im}(z_c)\, i^2 \\ &= \operatorname{Re}(z_v)\operatorname{Re}(z_c) + \operatorname{Re}(z_v)\operatorname{Im}(z_c)\, i + \operatorname{Im}(z_v)\operatorname{Re}(z_c)\, i - \operatorname{Im}(z_v)\operatorname{Im}(z_c) \\ &= (\operatorname{Re}(z_v)\operatorname{Re}(z_c) - \operatorname{Im}(z_v)\operatorname{Im}(z_c)) + (\operatorname{Re}(z_v)\operatorname{Im}(z_c) + \operatorname{Im}(z_v)\operatorname{Re}(z_c))\, i \end{aligned} \tag{4.12}$$

$$\begin{aligned} \operatorname{Re}(z_v z_c) &= \operatorname{Re}(z_v)\operatorname{Re}(z_c) - \operatorname{Im}(z_v)\operatorname{Im}(z_c) \\ \operatorname{Im}(z_v z_c) &= \operatorname{Re}(z_v)\operatorname{Im}(z_c) + \operatorname{Im}(z_v)\operatorname{Re}(z_c) \end{aligned} \tag{4.13}$$

As can be seen, a multiplication of a complex number by a complex number requires four multiplications and two additions. When performing this with operations on matrices, the minus above is a problem, as this cannot be performed by a sum. However, we can resolve this by changing the imaginary part of z_c to its negative value, resulting in $\operatorname{Im}'(z_c) = -\operatorname{Im}(z_c)$

$$\begin{aligned} \operatorname{Re}(z_v z_c) &= \operatorname{Re}(z_v)\operatorname{Re}(z_c) + \operatorname{Im}(z_v)\operatorname{Im}'(z_c) \\ \operatorname{Im}(z_v z_c) &= \operatorname{Re}(z_v)\operatorname{Im}(z_c) + \operatorname{Im}(z_v)\operatorname{Re}(z_c) \end{aligned} \tag{4.14}$$

When z_c is a constant, $\operatorname{Im}'(z_c)$ can be precomputed. Now that there is a clear path how to perform complex by complex multiplication, as if they were real values, the next step is to get equation (4.10) to work with the interleaved values $\tilde{x}$.

In equation (4.14), we can see that both the real and imaginary parts of the output depend on all four parts of the input values. Therefore, the sum from the DFT equation must sum over both the real and imaginary parts of the interleaved inputs $\tilde{x}$ to be able to compute the output value. Just as with the polyphase FIR filter, the amount of stored values equals $Q = 2P$, resulting in the following equation:

$$\tilde{X}_j = \sum_{q=0}^{Q-1} \tilde{x}(q)\tilde{\mathbf{W}}(j,q) \tag{4.15}$$

Where $\tilde{X}_j$ for all $j \in \{0, \ldots, Q-1\}$ represents the interleaved output with j even the real parts and j odd the imaginary parts. The interleaved input is $\tilde{x}$ and the matrix $\tilde{\mathbf{W}}$ is in $\mathbb{R}^{Q\times Q}$.

After this, all that remains is to define the values of $\tilde{\mathbf{W}}$. We do this by first decomposing the twiddle factor W into its real and imaginary parts.

$$\begin{aligned} W^{kp} &= e^{2\pi ikp/P} \\ &= \cos(2\pi kp/P) + i\ \sin(2\pi kp/P) \end{aligned} \tag{4.16}$$

$$\begin{aligned} \mathrm{Re}(W^{kp}) &= \cos(2\pi kp/P) \\ \mathrm{Im}(W^{kp}) &= \sin(2\pi kp/P) \end{aligned} \tag{4.17}$$

The sum in equation (4.15) is taken over every input sample multiplied by a constant value. Equation (4.14) gives us these values. Here, z_c are the constants W^{kp} and z_v are the inputs in $x(p)$. For clarity, we proceed with $\tilde{X}_j$ for j even and odd separately, we first look at j even, which are the real parts of the output.

With even values $j \in \{0, 2, \ldots, Q-2\}$, the equation for computing a real result is $\mathrm{Re}(z_v)\,\mathrm{Re}(z_c) + \mathrm{Im}(z_v)\,\mathrm{Im}'(z_c)$, which for this case equals $\mathrm{Re}(x(p))\,\mathrm{Re}(W^{kp}) + \mathrm{Im}(x(p))\,\mathrm{Im}'(W^{kp})$. Starting with the left part of this expression, the real part of $x(p)$ is stored at $\tilde{x}(2p)$. Meaning, we have to set $\tilde{\mathbf{W}}(j, 2p) = \mathrm{Re}(W^{kp})$ with $k = j/2$. Now for the right part, $\mathrm{Im}(x(p))$ is stored at $\tilde{x}(2p+1)$, resulting in $\tilde{\mathbf{W}}(j, 2p+1) = \mathrm{Im}'(W^{kp}) = -\mathrm{Im}(W^{kp})$ again with $k = j/2$. The index $2p+1$ is used because the matrix $\tilde{\mathbf{H}}$ must also be interleaved to be able to store the real and imaginary parts.

Now, the same logic is applied for odd values $j \in \{1, 3, \ldots, Q-1\}$. The expression for the imaginary part of a complex multiplication is $\mathrm{Re}(z_v)\,\mathrm{Im}(z_c) + \mathrm{Im}(z_v)\,\mathrm{Re}(z_c)$, which in this case equals $\mathrm{Re}(x(p))\,\mathrm{Im}(W^{kp}) + \mathrm{Im}(x(p))\,\mathrm{Re}(W^{kp})$. Let us again first focus on the left part. The real part of $x(p)$ is stored at $\tilde{x}(2p)$, so we set $\tilde{\mathbf{W}}(j, 2p) = \mathrm{Im}(W^{kp})$ with $k = (j-1)/2$. Now for the right part, $\mathrm{Im}(x(p))$ is stored at $\tilde{x}(2p+1)$, so we set $\tilde{\mathbf{W}}(j, 2p+1) = \mathrm{Re}(W^{kp})$ with $k = (j-1)/2$.

The logic for creating this DFT weight matrix is implemented in a PyTorch as a helper file named `dft_helper.py` in the accompanying repository.

With the weight matrix $\tilde{\mathbf{W}}$ defined, we are left with a valid vector matrix multiplication that must be mapped to neural network operations. Following the same reasoning used for the polyphase FIR filter implementation, we want to map this operation to a CNN. A vector matrix multiplication can be done with pointwise convolution, we again start with the equation for convolution:

$$s(i,j,c_2) = \sum_{c_1 \in \{C_{in}\}} \sum_{k,l \in \mathcal{N}} w(k,l,c_1,c_2)x(i+k,j+l,c_1) + b(c_2) \tag{4.18}$$

Pointwise convolution is performed by setting the kernel size to $(1,1)$, which also equals the last two input dimensions, so we only generate one output per c_2. This means that the inner loop is discarded and also the first and second output dimensions i and j. Since a bias is not required, b is also discarded, resulting in:

$$s(c_2) = \sum_{c_1 \in \{C_{in}\}} w(c_1,c_2)x(c_1) \tag{4.19}$$

We reverse the theoretical indexing of w into the real indexing:

$$s(c_2) = \sum_{c_1 \in \{C_{in}\}} w(c_2,c_1)x(c_1) \tag{4.20}$$

Since c_1 represent the inputs and c_2 the outputs, we rewrite them to match the in- and outputs of the DFT equation.

$$s(j) = \sum_{q=0}^{Q-1} w(j,q)x(q) \tag{4.21}$$

This matches the DFT equation (4.10) with $w(j,q) = \tilde{\mathbf{W}}(j,q)$ and $x(q) = \tilde{x}(q)$ with the output $s(j) = \tilde{X}(j)$.

In conclusion, a complex-to-complex DFT with interleaved values can also be effectively mapped to a CNN. The weight matrix generation is explained above and must be precomputed and stored with the CNN. NVDLA requires the input and output format to be of shape $\mathbb{R}^{N \times C \times H \times W}$, and sice pointwise convolution is used, values are stored in the C axis, where $C = Q = 2P$. Both H and W are set to 1. This DFT is implemented in `dft_cnn_module.py` with PyTorch. It supports all sizes, not just limited to powers of two.

For NVDLA, axis C is limited to 8192 values, so the maximum size of DFT that is supported is 4096. An implementation is visualized for a 256 point DFT with batch size of 5 in Figure 9.

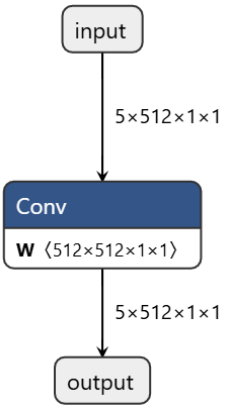

Figure 9: A 256 point complex-to-complex DFT with a batch size of 5. This model uses one convolution to perform the DFT. Image generated with Netron.

As for the complexity of this implementation. For a DFT with complex values P, this implementation performs $2P \cdot 2P$ multiplications, which gives a complexity of $4P^2 \in O(P^2)$

4.4 FFT

Implementing an FFT is more complex than implementing a DFT. This is largely due to the fact that an FFT can be seen as several layers, with complicated access patterns in between layers.

In this section, we first sketch a design for what must be computed in each layer, followed by a description of how the data must be reordered between the layers. We then implement this design with neural network operations that work on NVDLA.

4.4.1 Design

In Section 2.3.3, we saw that we could perform a P-point DFT by splitting the DFT into two smaller DFTs of $P/2$ points, which can be applied recursively. The butterfly operation was also shown. We recall equations (2.8) and (2.9) that together form a butterfly operation:

$$\begin{aligned} X_k &= X_k^e + W^k X_k^o \\ X_{k+P/2} &= X_k^e - W^k X_k^o \end{aligned} \tag{4.22}$$

As we later on want to map every operation to a neural network, we need to view the butterfly operation as an operation on a matrix. The equation above is equivalent to the following matrix vector multiplication:

$$\begin{bmatrix} X_k \\ X_{k+P/2} \end{bmatrix} = \begin{bmatrix} 1 & W^k \\ 1 & -W^k \end{bmatrix} \begin{bmatrix} X_k^e \\ X_k^o \end{bmatrix} \tag{4.23}$$

This performs the FFT for points k and $k + P/2$. This has to be performed for all $k \in \{1, \ldots, P/2\}$ to get all the outputs for $X \in \mathbb{R}^P$. This can be achieved by performing equation (4.23) in parallel on the input data or previous layers. Note that this is not one large matrix vector operation, but $P/2$ smaller matrix vector multiplications, each performing one butterfly operation.

As explained in Section 2.3.3, before the execution of the parallel butterfly operations (also called a layer), the order of the input data has to be in its bit-reverse order of the original index. After this layer, the output is in the form of $X_1, X_{P/2}, X_2, X_{P/2+1}, \ldots, X_P$. which is not in the correct order to be used as an input for the next layer. This can be seen by looking at the equation above. The butterfly requires the k-th index of the two smaller FFTs.

This is where reordering between butterfly layers comes in. We match the requirements for the next layer with the output from the previous layer. We also need one reordering at the end, because our final layer also produces an output as described above.

4.4.2 Implementation

As for the implementation, we again need to ensure that this works with our interleaved representation for complex values. We use the same way of interleaving the data in our input $\tilde{\mathbf{x}} \in \mathbb{R}^Q$ with $Q = 2P$ for a P-point complex to complex FFT.

We first focus on the butterfly layers. We need to perform $P/2$ butterflies per layer in parallel. This means $P/2$ complex-valued matrix vector multiplications. Each separate butterfly has input in $\mathbb{C}^2$, meaning it has an interleaved input in $\mathbb{R}^4$. It also generates an output in $\mathbb{R}^4$, so requires an interleaved weight matrix in $\mathbb{R}^{4\times4}$ In Section 4.3.2, we show how to perform the decomposition of a complex value multiplication. We also show how to generate the interleaved matrix $\tilde{\mathbf{W}}$ of twiddle factors. The weight matrix required for the FFT butterfly operation is just a special case of the normal twiddle factor matrix, so this creation will not be discussed again, as the same reasoning can be used.

Next, we show how to perform $P/2$ matrix multiplications in parallel with a CNN, starting with its mathematical representation 2.11:

$$s(i,j,c_2) = \sum_{c_1 \in \{C_{in}\}} \sum_{k,l \in \mathcal{N}} w(k,l,c_1,c_2)x(i+k,j+l,c_1) + b(c_2) \tag{4.24}$$

We first discard the dimensions indexed by i and j, drop the bias addition and set the kernel size to 1×1:

$$s(c_2) = \sum_{c_1 \in \{C_{in}\}} w(c_1,c_2)x(c_1) \tag{4.25}$$

Next, we reason about how to perform the $P/2$ separate operations. Let us look at pointwise convolution. One would use this if no interaction along the channel axis is required. This is done by setting the number of groups equal to the number of channels. If we set the number of groups to half the number of channels, then there is interaction between pairs of channels. This effectively creates $C/2$ separate operations and this is exactly what we require. Because we are working with interleaved values, we actually require $C/4$ groups, as four channels have to be paired to perform the interleaved matrix vector multiplication of a butterfly. This means setting the amount of groups equal to $P/2$ or $Q/4$.

With the butterfly multiplication mapped on a CNN, we require a way to reorder the data. The reordering of data has to be done before the first layer, between each layer and after the last layer. We can use a split layer to split the data, along an axis. This can be followed by a concat layer to reconstruct the data in a required order. In split we can specify how large each part of the split must be. For this to efficiently work with

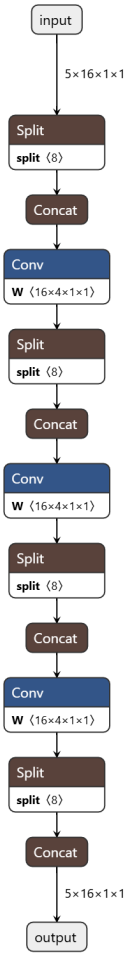

Figure 10: A visualization of an 8 point FFT with 3 layers and a batch size of 5. Image generated with Netron.

the interleaved representation, we can set the size to 2, as the real and imaginary parts are always kept next to each other.

In conclusion, we have shown how to perform the butterfly multiplications in parallel using a CNN with groups set equal to $P/2$ and a kernel of size 1×1. The weight matrix is a special case of the DFT weight matrix. We reorder the data in between layers with split and concat. The input and output dimensions are the same as with the DFT implementation, they are in $\mathbb{R}^{N \times Q \times 1 \times 1}$ As each layer performs $4Q = 8P$ multiplications and there are $\lg P$ layers, this is performed in a complexity of $8P \lg P \in O(P \log P)$. We have also shown how to perform the reordering of the data between each layer using split and concat operations. This FFT is implemented in the `fft_cnn_module.py` and a visual representation for a 8-point FFT with a batch size of 5 can be seen in Figure 10.

5 Experiments

Part of this paper is an investigation of the performance of the implemented polyphase filter bank. This is so that we can determine whether a PFB implemented on NVDLA can compete, in energy used and performance, to a GPU PFB. We do this by benchmarking the PFB implementations and comparing it to the performance of a GPU reference implementation developed for LOFAR.

5.1 Setup

The benchmark was performed on a NVIDIA Jetson Orin AGX 64GB [2], which draws between 8 and 65 watts. The Jetson has two NVDLA v2 accelerators. The NVDLA accelerators make up 105 out of the 275 INT8 tops available on the Jetson AGX Orin. The memory bandwidth of the Jetson is 204.8 GB/s.

We used the PowerSensor3 board with the USB-C connector to measure the amount of joules that were being used. The PowerSensor3 measures the total power consumption of the whole system.

The following software versions were used:
NVIDIA JetPack 6.1 (CUDA 12.6, g++ 10.4)
TensorRT 10.7 - installed separately, as TensorRT 10.3 contained in JetPack 6.1 contains a bug.
PowerSensor3 v1.6.1
Python 3.10, with PyTorch 2.6.0, ONNX 1.17 and NumPy 1.26.

5.1.1 Model Generation Procedure

The accompanying repository supports generating models of any size, only limited by hardware. The `pytorch_model/main.py` script provides a CLI to create most configurations by default and has an option to create all benchmarking files. The supported models are pfb_dft, pfb_fft, fir, dft, fft. The code can be easily modified for more customization.

The generation of models works in two stages. First, a PyTorch model is converted to ONNX. This ONNX file is then compiled with TensorRT `trtexec` to create a NVDLA loadable. When this is done, there are two options for inference provided in the `runtime` folder. There is a benchmarking application named `benchmark` and a simple inference application called `simple`. The benchmarking application runs a benchmarking suite for a whole folder of NVDLA models, more on this in the

following section. The simple application is meant for testing and runs inference once. It supports reading and writing of in- and output data to files.

5.2 Methodology

This section reveals the experiments that were performed to evaluate the implementation on NVDLA. As the current LOFAR pipeline uses a PFB of 256 channels with 16 taps, we benchmarked these configurations on NVDLA. In the first two sections, we describe two different groups of models that were compared, followed by a section on how the benchmark was technically done. In the last section, we discuss how we measured the accuracy.

5.2.1 DFT and FFT Polyphase Filter Bank

The main research question focuses on the entire polyphase filter bank. Both a DFT and FFT variants were implemented, both of which required benchmarking. Since the amount of taps and channels is fixed, the only thing we vary is the batch size. We ran the experiment for the following batch sizes: $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096$.

When building the PFB models for these batch sizes, `trtexec` runs out of memory for the FFT model with a batch size larger than 32. For this reason, the FFT version is only tested with a batch size up to 32.

5.2.2 Polyphase FIR filter and DFT Separately

Secondly, we are interested in how different parts of the PFB perform. We again ran the benchmark procedure for 256 channels and 16 taps, but this time for the separate FIR and DFT models. Both models were tested with the following batch sizes: $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096$.

5.2.3 Benchmarking procedure

The exact benchmarking code can be found in the `runtime` folder of the repository. The code contains a few wrappers and lifecycle managers around the CUDA and cuDLA API for ease of use and some code to perform the benchmark. The benchmark itself consists of a few stages and can be explained with the pseudocode in Appendix A.

The benchmark is executed for 20 iterations, interleaving the different models. For each iteration of a model, the model is executed multiple times with a warm-up phase. Before the warm-up phase, enough buffers are made to ensure that when a buffer is used again, it is no longer held in cache or SRAM. We also measured the base power draw before each warm-up phase, so we could estimate the base energy consumption of the system during the execution of the model. Since we aimed to compute about the same amount of samples each run, and a bigger batch size means more samples per invocation, the number of invocations differs per batch size. The execution of PowerSensor3 measurements was performed with host callback functions inserted into the same CUDA stream that was used by the NVDLA, this way, accurate timing could be achieved. The built-in method of the PowerSensor3 library was utilized to calculate the joules and average watts used between two measurements. A last very important note is that only one DLA was used at a time.

After running the benchmark, we took averages over the iterations for each model. We also computed metrics such as DLA energy efficiency (samples per joule) and bandwidth. The benchmark estimates the base energy consumption by measuring the power draw before the model is executed. Next, this power draw is multiplied by the time it took the model to execute. The estimated base energy is then subtracted from the total energy, resulting in the DLA energy consumption.

Note that the bandwidth is a theoretical number, not the actual bandwidth to and from NVDLA. The bandwidth used for the weights is not included in this number. The reported bandwidth equals the number of input samples (with the number of historical samples) plus the number of output samples expressed in MB/s. The analysis code can be found in `analysis/analysis.ipynb`.

5.2.4 Accuracy

Due to the fact that NVDLA only works with FP16 or INT8, a drop in accuracy is expected compared to the reference implementation. To review this, we run an NVDLA DFT with batch size 512 with the `simple` inference application. We provide it with an input file with random FP16 values between 0 and 100. We then compare the results with the PyTorch `torch.fft.fft` function, which performs an FFT in FP32. We report on the maximum absolute error and the mean absolute error found. The script for this analysis can be found in `accuracy.ipynb`.

5.3 Results

In this section, we present the results that we received from the experiments described in the previous section. We first list the results from the benchmarks, followed by the results from our accuracy comparison.

5.3.1 DFT and FFT Polyphase Filter Bank

The polyphase filter bank was tested for 256 complex-valued channels and 16 taps with varying batch sizes. We tested both the DFT and FFT versions.

Figure 11 contains two graphs. The left graph is the PFB version with an DFT, the right version uses an FFT. In each graph, the left axis shows the throughput in samples per second and the right axis shows the efficiency in joules per second. On the horizontal axis, the different batch sizes are shown. Notice that the left axis of the two graphs does not use the same scale, the DFT version shows higher throughput and efficiency.

In addition to the figure above, we also report Figure 12, which shows the power draw in watts for different models. As base power draw is an important measurement that could skew our results if done improperly, we also include a graph of the estimated base power draw in Figure 13. The average base power draw during the PFB experiment was 11.22 ± 0.17 watts.

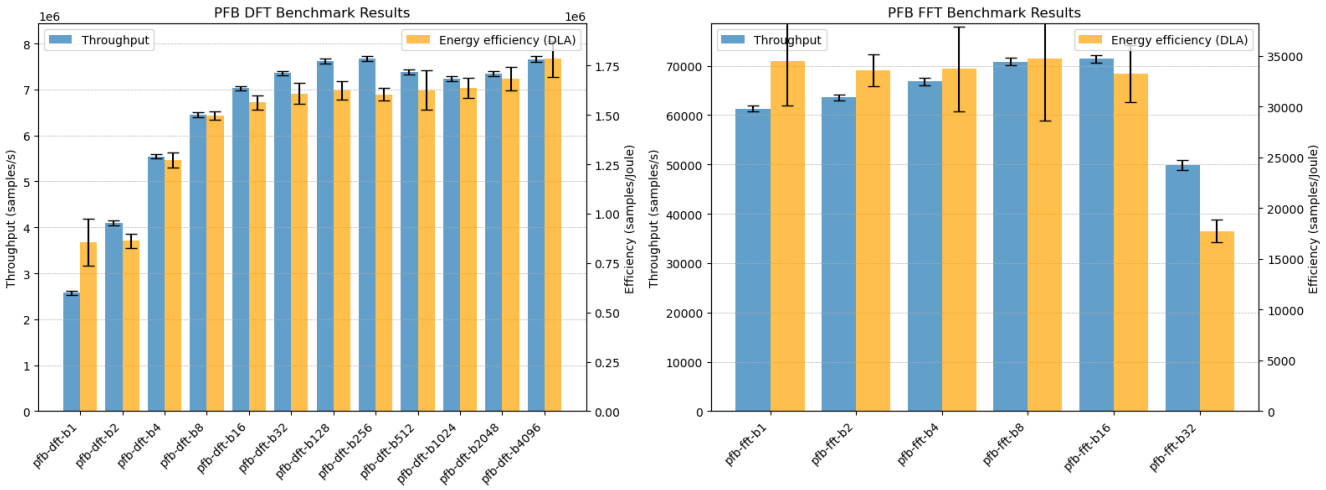

Figure 11: A comparison of the two PFB implementations. With the DFT version on the left and the FFT version on the right. The left axis shows the throughput of the models and the right axis shows the efficiency of said model.

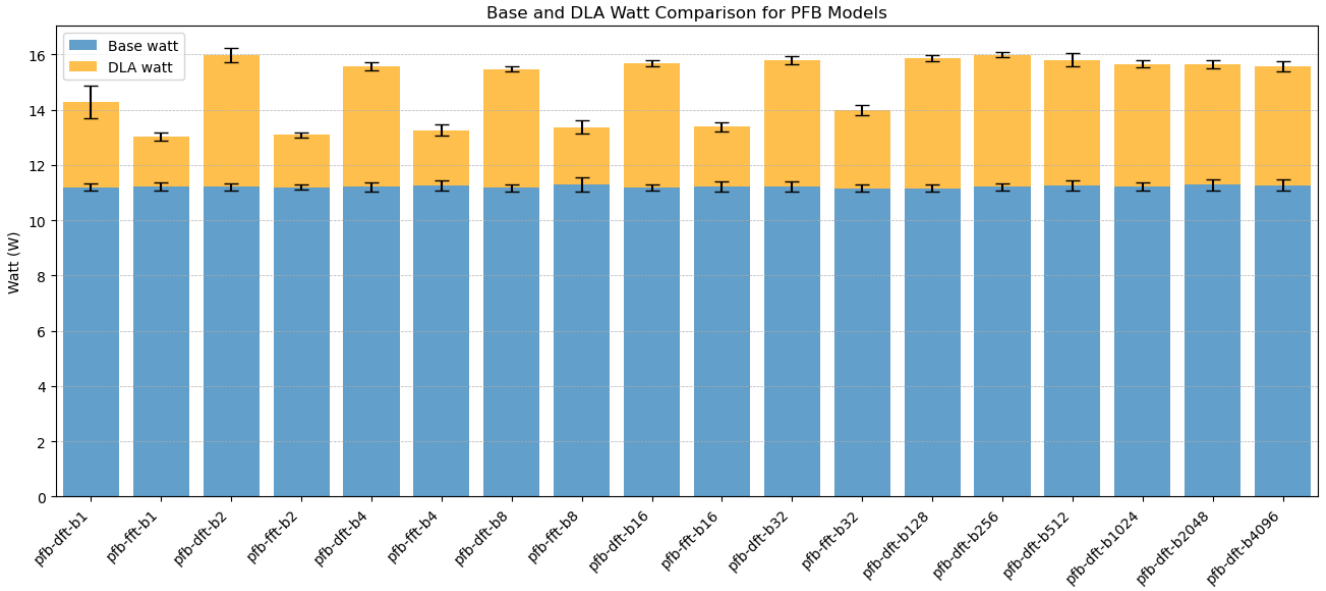

Figure 12: A comparison of power draw for the NVDLA DFT and FFT PFB, ordered from small batch size, to large.

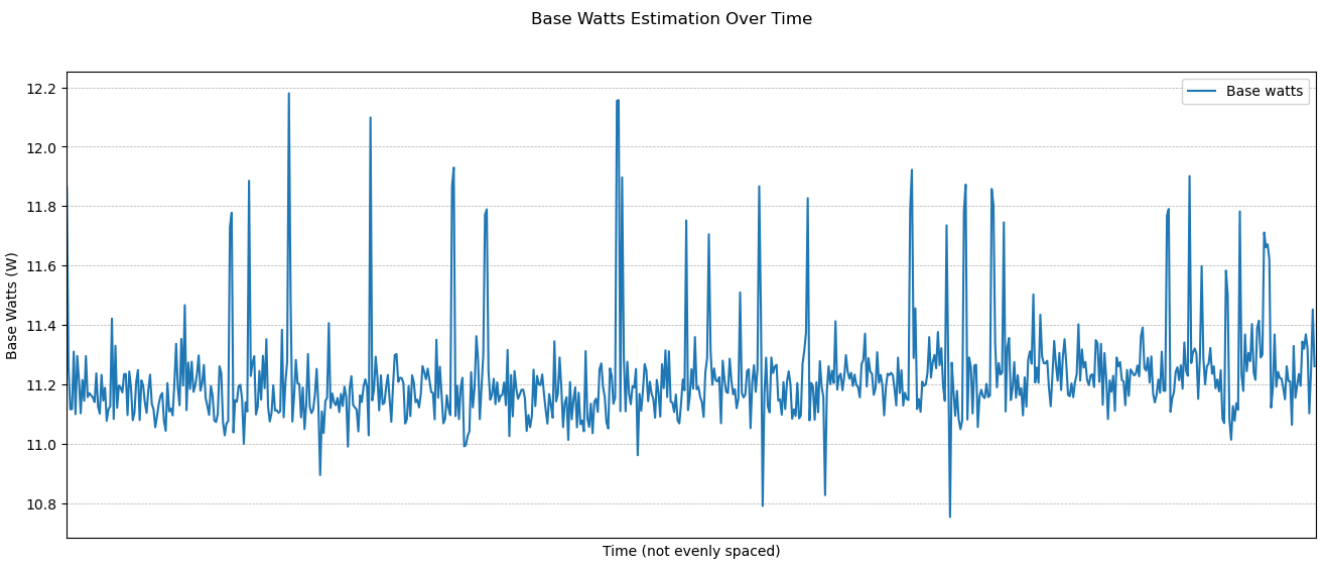

Figure 13: The base power estimation during the execution of the PFB benchmark, as calculated by the benchmarking application. The mean is 11.22 ± 0.17 watts. The values are not evenly sampled in time, due to the fact that execution times differs per NVDLA model.

In Table 1 of Appendix B, we list additional results that are not present in the graphs. Here we list the number of samples $\times 10^6$ used to obtain the results, as well as the bandwidth achieved. Note again that this is not the real bandwidth, only a calculation and that the weight data is not included in this.

5.3.2 Polyphase FIR filter and DFT Separately

For the experiment with the polyphase FIR filter and DFT separately, we again tested a 256 channel DFT with 16 taps for the FIR. The throughput and efficiency are listed in Figure 14. In this figure, we have the DFT results on the left and the FIR results on the right, with the left axis representing the throughput and the right axis representing the efficiency.

In Table 2 of Appendix B we list the same additional results. When comparing the additional results with the additional results from the PFB experiment, we notice that the bandwidth is highest with just the FIR model.

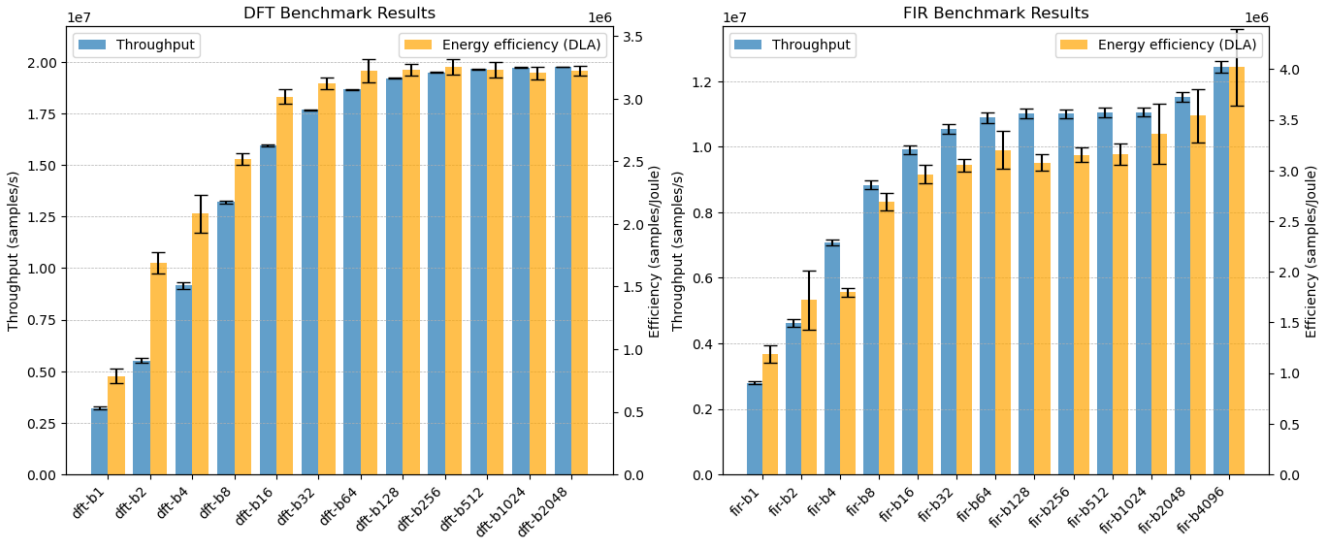

Figure 14: The results of an experiment comparing separate parts of a PFB. On the left we have the results for a DFT and on the right the results for a polyphase FIR filter. The vertical axis show the results, with the left axis the throughput in samples per second and on the right the energy efficiency in samples per joule.

5.3.3 Accuracy

When comparing the accuracy of the 256-point NVDLA DFT with PyTorch as a reference, we obtain a maximum absolute error of 4.2 and a mean absolute error of 0.19. The input values where random values between 0 and 100.

6 Discussion

In the following section, we discuss and interpret the results we obtained from testing the implementation of a PFB and the separate parts of the PFB. We compare the results with an implementation for LOFAR. We do this in sections related to our three sub-questions, which are: *What is the difference in throughput between the NVDLA and a CUDA kernel?* Secondly: *What is the difference in throughput per joule between the CUDA kernel and NVDLA?* And lastly: *What is the difference in accuracy between a CUDA kernel and NVDLA?*

After discussion of our sub-questions, we shortly discuss the broader implications of our results.

6.1 Throughput Comparison

We begin by analysing the throughput of the DFT PFB implementation and the FFT PFB implementation. Figure 11 shows that the throughput in samples per second for the DFT PFB reaches approximately $7.6 \cdot 10^6$ for the better-performing batch sizes. For the FFT PFB version, this is approximately $7.1 \cdot 10^4$ samples per second. This means that the DFT version is about two orders of magnitude faster. This is somewhat surprising, as an FFT is an optimization of a DFT. A DFT has a complexity of $O(n^2)$ and an FFT has a complexity of $O(n \log n)$. One working theory is

that the RUBIK module, described in Section 2.7.1 processes the re-order operations between the FFT layers much slower than the additional multiplications take to compute, as required by the DFT.

To compare NVDLA PFB throughput with a CUDA implementation for LOFAR, we take the throughput given in Section 3.3. These statistics were taken on the same hardware and equal to about $6.7 \cdot 10^9$ samples per second at a data rate of 80.1 GB/s. This is 3 orders of magnitude faster than the DFT version and 5 than the FFT version. The peak data rate of the NVDLA DFT PFB version can be found in Table 1 in Appendix B and equals 521 ± 3.57 MB/s. Recall that this is a calculated data rate; nonetheless, this is also much lower than the data rate of 80.1 GB/s for the CUDA version. Although the results were taken with only one NVDLA in use, utilizing both NVDLAs could only double the throughput.

As the throughput of the full PFB on NVDLA differs 3 orders of magnitude with the CUDA implementation, we also investigated the throughput that the separate parts of the PFB could achieve. These results are shown in Figure 14. The DFT alone can achieve about $2 \cdot 10^7$ samples per second and a polyphase FIR filter $1.2 \cdot 10^7$. While this is faster than the whole PFB, this is still too slow to compete with the GPU implementation.

In short, NVDLA cannot compete the throughput of a CUDA kernel. Even the separate parts of are two orders of magnitude too slow on NVDLA to be used to for offloading certain parts of the CUDA implementation.

6.2 Energy Efficiency

The energy efficiency of the PFB can be seen in Figure 11. This energy usage is the energy consumed by the whole system, from which an estimated base energy is subtracted. We can see that energy efficiency is closely related to throughput.

Since the CUDA implementation has a throughput orders of magnitude higher than the NVDLA PFB, it would mean that in order for the NVDLA version to compete in energy efficiency, the CUDA version has to consume an order of 3 magnitudes more than the NVDLA PFB. As can be seen in Figure 12, the DLA consumes at least 3 watts when the NVDLA model is being executed. This would mean that the CUDA version would have to draw more than $3 \cdot 10^3 = 3000$ watts to be more inefficient than the NVDLA PFB. As the system can draw 65 watts maximum, this is not possible.

The energy usage of the separate parts of the PFB were not compared to a CUDA implementation, as there is no CUDA implementation for just the FIR or FFT parts to reference against. But the same argumentation as above can be repeated, making it unlikely that it will be able to compete on efficiency.

In conclusion, because of the shear difference in throughput between the CUDA and NVDLA versions, the theoretically energy efficient NVDLA cannot compete in efficiency in the case of a PFB.

6.3 Accuracy Comparison

An absolute maximum error of 4.2 for input values between 0 and 100 is high. There are many applications that suffer from such an inaccuracy. This means a PFB implementation cannot just be replaced with a NVDLA version, the repercussions have to be analysed for such a drop in accuracy.

6.4 Broader Implications

From the throughput comparison, we see that a standard optimization for an algorithm may not always also result in better results on NVDLA. The DFT outperforms the FFT by two orders of magnitude. Why this is, is not known to us. It might be because the RUBIK module of NVDLA is much slower than the convolution pipeline. Or because of some unseen side effect of using smaller convolutions in the FFT implementation. In short, what is faster on a CPU/GPU might not be faster on NVDLA, so extensive testing has to be done before choosing an implementation.

NVIDIA claims that the NVDLAs make up 105 of the 275 INT8 tops on the Jetson Orin AGX. No statements about FP16 throughput could be found, but from the results in throughput for a PFB, it does not seem to be anywhere near the almost 40% it is for INT8.

6.5 Limitations

There are several limitations in this paper. First and foremost, the TensorRT compiler functions completely as a black box when turning an ONNX model into an NVDLA loadable. If the TensorRT compiler does not perform this effectively, the results might show worse performance than what could have been achieved. This might have been solved with further investigation of the loadable file and the open-source compiler, but time was a factor.

Secondly, nothing could be found on what exact specifications of the NVDLA are in the Jetson Orin AGX. The hardware standard is open, but there are still many options that are configurable and could give insight into how to create a performant model.

This paper also does not take into account the latency that NVDLA might introduce with a large batch size. If the processing time for a larger batch size is long, it will produce results in bursts. This would probably require some form of buffering, increasing the integration complexity and possibly performance.

Lastly, the benchmark was run on only one NVDLA at a time, while there are two NVDLAs available.

7 Future work

An interesting topic to investigate is the throughput that an INT8 implementation could produce. NVDLA loadables with the INT8 data type can be produced with `trtexec` and the runtime application would require minimal modification. INT8 would imply less accuracy, so its use case is more limited. Whether or not the 105 INT8 tops can be reached with an INT8 variant can also shed some light on the FP16 tops by comparison.

The current FP16 implementation could also be tested for a PFB with more channels or more taps. In its current form, the model can be compiled for NVDLA with up to 32 taps and 4096 complex channels. This would not directly fit the requirements of LOFAR, but there might be some use for it.

As a third possibility, we look at the trend in throughput of the results for different batch sizes. One can notice that the performance levels off, but increases again for the largest batch sizes. It is a possibility that the NVDLA requires more work at the same time to reach a higher throughput. Instead of just relying on the batch dimension, it would also be possible to perform two polyphase filter banks side by side. This would mean that the weights for the CNNs have to be transferred twice, once for each PFB, but the increase in parallelism might justify this.

Lastly, since this PFB is made with PyTorch and consists of CNNs, it should be transferable to other deep learning accelerators. Researching the throughput on those devices might give some interesting results.

8 Conclusion

This paper aims to investigate the effectiveness of a polyphase filter bank on NVDLA and its comparison to a CUDA implementation. It does so by first implementing a PFB and then comparing its performance.

Three main models are implemented. A polyphase FIR filter, a DFT and an FFT. These implementations are performed in two steps. First, a design is presented for each part. This design is a mathematical representation of the model as operations that need to be performed on a matrix. In the second step, the design is implemented with neural network operations.

For the polyphase FIR filter, we show that it can be mapped to a CNN with a kernel size equal to the number of taps and a group size equal to the number of channels, also called depthwise convolution. Next, we show that a DFT can also be implemented using a CNN using pointwise convolution, meaning a kernel size of 1. We also pay special attention to the creation of the weight matrix required by the DFT for handling complex numbers. Lastly we show how we can implement an FFT. We do this by showing how split and concat layers can be used to re-order the data, and again use CNNs to perform multiplications.

With the models implemented, we benchmark them on a Jetson Orin AGX 64GB using 16 bit floating-point (FP16) precision. We chose 256 complex-valued channels and 16 taps, as this is the configuration used at LOFAR. The results show that the DFT PFB implementation has a throughput of about $7.6 \cdot 10^6$ samples/s and the FFT PFB approximately $7.1 \cdot 10^4$ samples/s. However, the LOFAR CUDA implementation can do $6.7 \cdot 10^9$, answering our first sub-question *How does a polyphase filter implemented in NVDLA perform compared to a CUDA implementation?*. The second sub-question is *What is the difference in throughput per joule between the CUDA kernel and NVDLA?*. The conclusion to this question is also based on the throughput of the CUDA implementation. We conclude that three orders of magnitude difference in throughput cannot be compensated for by the low power draw of NVDLA. As for our last sub-question, *What is the difference in accuracy between a CUDA kernel and NVDLA?*, we see a maximum absolute error of 4.1 for our DFT with input values ranging between 0 and 100. Limiting its general applicability.

In summary and answering the main research question, *How does a polyphase filter implemented in NVDLA perform compared to a CUDA implementation?*. It cannot compete with a CUDA GPU implementation, both in throughput and efficiency. The lower accuracy is also something one would have to accept.

References

[1] K. van der Veldt, R. van Nieuwpoort, A. L. Varbanescu, and C. Jesshope, "A polyphase filter for GPUs and multi-core processors," in *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*, 2012, pp. 33–40.

[2] L. S. Karumbunathan, "NVIDIA Jetson AGX Orin Series - A Giant Leap Forward for Robotics and Edge AI Applications," Jul. 2022. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf

[3] B. Ryden and B. M. Peterson, *Astronomical Detection of Light*. Cambridge University Press, 2020.

[4] M. P. van Haarlem, M. W. Wise, A. W. Gunst, G. Heald, J. P. McKean, J. W. T. Hessels, A. G. de Bruyn, R. Nijboer, J. Swinbank, R. Fallows, M. Brentjens, A. Nelles, R. Beck, H. Falcke, R. Fender, J. Hörandel, L. V. E. Koopmans, G. Mann, G. Miley, H. Röttgering, B. W. Stappers, R. A. M. J. Wijers, S. Zaroubi, M. van den Akker, A. Alexov, J. Anderson, K. Anderson, A. van Ardenne, M. Arts, A. Asgekar, I. M. Avruch, F. Batejat, L. Bähren, M. E. Bell, M. R. Bell, I. van Bemmel, P. Bennema, M. J. Bentum, G. Bernardi, P. Best, L. Bîrzan, A. Bonafede, A.-J. Boonstra, R. Braun, J. Bregman, F. Breitling, R. H. van de Brink, J. Broderick, P. C. Broekema, W. N. Brouw, M. Brüggen, H. R. Butcher, W. van Cappellen, B. Ciardi, T. Coenen, J. Conway, A. Coolen, A. Corstanje, S. Damstra, O. Davies, A. T. Deller, R.-J. Dettmar, G. van Diepen, K. Dijkstra, P. Donker, A. Doorduin, J. Dromer, M. Drost, A. van Duin, J. Eislöffel, J. van Enst, C. Ferrari, W. Frieswijk, H. Gankema, M. A. Garrett, F. de Gasperin, M. Gerbers, E. de Geus, J.-M. Grießmeier, T. Grit, P. Gruppen, J. P. Hamaker, T. Hassall, M. Hoeft, H. A. Holties, A. Horneffer, A. van der Horst, A. van Houwelingen, A. Huijgen, M. Iacobelli, H. Intema, N. Jackson, V. Jelic, A. de Jong, E. Juette, D. Kant, A. Karastergiou, A. Koers, H. Kollen, V. I. Kondratiev, E. Kooistra, Y. Koopman, A. Koster, M. Kuniyoshi, M. Kramer, G. Kuper, P. Lambropoulos, C. Law, J. van Leeuwen, J. Lemaitre, M. Loose, P. Maat, G. Macario, S. Markoff, J. Masters, R. A. McFadden, D. McKay-Bukowski, H. Meijering, H. Meulman, M. Mevius, E. Middelberg, R. Millenaar, J. C. A. Miller-Jones, R. N. Mohan, J. D. Mol, J. Morawietz, R. Morganti, D. D. Mulcahy, E. Mulder, H. Munk, L. Nieuwenhuis, R. van Nieuwpoort, J. E. Noordam, M. Norden, A. Noutsos, A. R. Offringa, H. Olofsson, A. Omar, E. Orrú, R. Overeem, H. Paas, M. Pandey-Pommier, V. N. Pandey, R. Pizzo, A. Polatidis, D. Rafferty, S. Rawlings, W. Reich, J.-P. de Reijer, J. Reitsma, G. A. Renting, P. Riemers, E. Rol, J. W. Romein, J. Roosjen, M. Ruiter, A. Scaife, K. van der Schaaf, B. Scheers, P. Schellart, A. Schoenmakers, G. Schoonderbeek, M. Serylak, A. Shulevski, J. Sluman, O. Smirnov, C. Sobey, H. Spreeuw, M. Steinmetz, C. G. M. Sterks, H.-J. Stiepel, K. Stuurwold, M. Tagger, Y. Tang, C. Tasse, I. Thomas, S. Thoudam, M. C. Toribio, B. van der Tol, O. Usov, M. van Veelen, A.-J. van der Veen, S. ter Veen, J. P. W. Verbiest, R. Vermeulen, N. Vermaas, C. Vocks, C. Vogt, M. de Vos, E. van der Wal, R. van Weeren, H. Weggemans, P. Weltevrede, S. White, S. J. Wijnholds, T. Wilhelmsson, O. Wucknitz, S. Yatawatta, P. Zarka, A. Zensus, and J. van Zwieten, "LOFAR: The LOw-Frequency ARray," *Astronomy & Astrophysics*, vol. 556, p. A2, Jul. 2013, publisher: EDP Sciences. [Online]. Available: http://dx.doi.org/10.1051/0004-6361/201220873

[5] D. Nébouy, "Printing quality assessment by image processing and color prediction models," PhD Thesis, Dec. 2015.

[6] D. C. Price, "Spectrometers and polyphase filterbanks in radio astronomy," in *The WSPC Handbook of Astronomical Instrumentation: Volume 1: Radio Astronomical Instrumentation*. World Scientific, 2021, pp. 159–179.

[7] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965, publisher: JSTOR.

[8] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[9] G. C. Danielson and C. Lanczos, "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids," *Journal of the Franklin Institute*, vol. 233, no. 5, pp. 435–452, 1942, publisher: Elsevier.

[10] R. Szeliski, *Computer vision: algorithms and applications*. Springer Nature, 2022.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, publisher: Ieee.

[12] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf

[13] ONNX, "ONNX 1.19.0 documentation." [Online]. Available: https://onnx.ai/onnx/index.html

[14] NVIDIA, "TensorRT SDK | NVIDIA Developer." [Online]. Available: https://developer.nvidia.com/tensorrt

[15] ——, "CUDA Toolkit | NVIDIA Developer." [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[16] NVIDIA, "NVIDIA Deep Learning Accelerator." [Online]. Available: https://nvdla.org/

[17] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[18] NVIDIA, "cuDLA API :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/cudla-api/

[19] S. v. d. Vlugt, L. Oostrum, G. Schoonderbeek, B. v. Werkhoven, B. Veenboer, K. Doekemeijer, and J. W. Romein, "PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool," 2025, _eprint: 2504.17883. [Online]. Available: https://arxiv.org/abs/2504.17883

[20] C. Boerkamp, S. v. d. Vlugt, and Z. Al-Ars, "TINA: Acceleration of Non-NN Signal Processing Algorithms Using NN Accelerators," 2024, _eprint: 2408.16551. [Online]. Available: https://arxiv.org/abs/2408.16551

[21] Olivier Visser, "Radio interference monitoring with low-end AI-hardware," Bsc Thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, Leiden, 2025.

[22] R. van Nieuwpoort, "polyphase filter bank generator." [Online]. Available: https://github.com/NLeSC/polyphase-filter-bank-generator

A Benchmark pseudocode

Algorithm 1 Main benchmarking procedure

```
1: stream ← createCudaStream
2: procedure benchmark
3:     // Run the experiment multiple iterations (20)
4:     for i = 1 to ITERATIONS do
5:         // Run the experiment for all the loadables.
6:         for m in loadables do
7:             // Run the experiment on both DLA's sequentially.
8:             for d = 0 up to 1 do
9:                 benchmarkSingle(m, d)
10:            end for
11:        end for
12:    end for
13: end procedure
```

Algorithm 2 Single benchmarking procedure

```
1: procedure BENCHMARKSINGLE(m, dla)
2:     //Load and initialize the NVDLA module
3:     INITMODULE(m, dla)
4:     nBuff, nRun ← CALCULATEBUFFERSANDRUNS(m)
5:     buffers ← INITBUFFERS(nBuff)
6:     // Allow everything to return to idle before we measure base energy consumption
7:     SLEEP(2s)
8:     // Measure base power consumption over 100ms
9:     MEASUREBASECONSUMPTION
10:    // The next steps are all enqueued async on the stream. The warm-up phase should help
       ensure there are no gaps in the execution of the actual experiment.
11:    for b = 1 to nBuff do // Warm-up
12:        ENQUEUEMODEL(stream, m, buffers[b])
13:    end for
14:    // We insert a start timer and a call back to start measuring energy usage.
15:    INSERTSTART(stream)
16:    // Enqueue the actual benchmark
17:    for r = 1 to nRun do
18:        for b = 1 to nBuff do
19:            ENQUEUEMODEL(stream, m, buffers[b])
20:        end for
21:    end for
22:    // We insert a stop timer and a call back to stop measuring energy usage.
23:    INSERTSTOP(stream)
24:    // Wait for the benchmark to finish.
25:    SYNCSTREAM(stream)
26:    // Write results to csv file.
27:    WRITERESULTS
28: end procedure
```

B Additional Results

Model	Msamples	Base Energy (J)	Total Energy (J)	Time (ms)	Bandwidth (MB/s)
pfb-dft-b1	10.182	44.3 ± 0.883	56.5 ± 1.52	3954 ± 77.0	175 ± 3.47
pfb-fft-b1	10.182	1864 ± 24.2	2162 ± 16.9	166009 ± 1448	4.17 ± 0.0364
pfb-dft-b2	10.182	27.9 ± 0.536	39.7 ± 0.359	2486 ± 36.3	279 ± 4.07
pfb-fft-b2	10.182	1794 ± 22.7	2098 ± 16.6	160162 ± 1376	4.32 ± 0.0371
pfb-dft-b4	10.182	20.6 ± 0.389	28.6 ± 0.292	1836 ± 15.1	377 ± 3.11
pfb-fft-b4	10.182	1715 ± 36.0	2020 ± 21.5	152271 ± 1751	4.55 ± 0.0523
pfb-dft-b8	10.199	17.7 ± 0.256	24.5 ± 0.238	1581 ± 12.0	439 ± 3.33
pfb-fft-b8	10.199	1626 ± 40.1	1925 ± 19.5	143961 ± 1547	4.82 ± 0.0518
pfb-dft-b16	10.199	16.2 ± 0.213	22.8 ± 0.229	1451 ± 10.3	478 ± 3.40
pfb-fft-b16	10.199	1603 ± 32.4	1912 ± 20.5	142835 ± 1592	4.86 ± 0.0541
pfb-dft-b32	10.224	15.6 ± 0.295	22.0 ± 0.192	1390 ± 9.84	500 ± 3.54
pfb-fft-b32	10.224	2290 ± 45.1	2869 ± 64.1	205031 ± 3957	3.39 ± 0.0655
pfb-dft-b128	10.224	15.0 ± 0.188	21.3 ± 0.134	1342 ± 9.15	518 ± 3.53
pfb-dft-b256	9.961	14.6 ± 0.196	20.8 ± 0.209	1299 ± 8.90	521 ± 3.57
pfb-dft-b512	9.437	14.4 ± 0.254	20.2 ± 0.244	1280 ± 9.58	501 ± 3.75
pfb-dft-b1024	8.389	13.0 ± 0.210	18.2 ± 0.137	1160 ± 9.01	492 ± 3.82
pfb-dft-b2048	8.389	12.9 ± 0.258	17.9 ± 0.187	1142 ± 9.08	499 ± 3.97
pfb-dft-b4096	8.389	12.4 ± 0.266	17.1 ± 0.168	1095 ± 8.40	521 ± 3.99

Table 1: Results from the PFB experiment comparing the DFT and FFT implementation. This table shows additional information not present in Figure 11.

Model	Msamples	Base Energy (J)	Total Energy (J)	Time (ms)	Bandwidth (MB/s)
dft-b1	9.999	35.0 ± 0.859	47.8 ± 1.48	3113 ± 67.4	25.7 ± 0.564
fir-b1	10.182	40.8 ± 0.943	49.4 ± 0.974	3641 ± 62.5	190 ± 3.30
dft-b2	9.999	20.4 ± 0.608	26.3 ± 0.620	1811 ± 41.8	44.2 ± 1.04
fir-b2	10.182	24.8 ± 0.930	30.9 ± 0.912	2195 ± 60.1	316 ± 8.75
fir-b4	10.182	16.2 ± 0.427	21.8 ± 0.240	1438 ± 15.4	481 ± 5.18
dft-b4	9.994	12.3 ± 0.261	17.1 ± 0.269	1092 ± 19.0	73.3 ± 1.24
dft-b8	9.994	8.50 ± 0.108	12.5 ± 0.102	757 ± 3.00	106 ± 0.418
fir-b8	10.199	13.0 ± 0.266	16.8 ± 0.215	1154 ± 16.3	601 ± 8.49
dft-b16	9.994	7.04 ± 0.112	10.3 ± 0.0937	627 ± 1.63	128 ± 0.331
fir-b16	10.199	11.5 ± 0.207	15.0 ± 0.218	1029 ± 14.7	674 ± 9.66
fir-b32	10.224	10.8 ± 0.205	14.2 ± 0.185	969 ± 14.1	718 ± 10.4
dft-b32	9.994	6.34 ± 0.0988	9.53 ± 0.0736	566 ± 1.14	141 ± 0.284
dft-b64	9.994	6.01 ± 0.0700	9.11 ± 0.0530	535 ± 0.746	149 ± 0.208
fir-b64	10.224	10.6 ± 0.229	13.8 ± 0.193	939 ± 13.7	741 ± 10.8
dft-b128	9.830	5.73 ± 0.0545	8.76 ± 0.0282	512 ± 0.538	154 ± 0.162
fir-b128	10.224	10.3 ± 0.209	13.7 ± 0.179	928 ± 12.9	750 ± 10.4
dft-b256	9.830	5.65 ± 0.0794	8.67 ± 0.0521	504 ± 0.445	156 ± 0.138
fir-b256	9.961	10.1 ± 0.148	13.3 ± 0.161	905 ± 11.4	749 ± 9.41
fir-b512	9.437	9.57 ± 0.0996	12.6 ± 0.149	854 ± 10.7	751 ± 9.40
dft-b512	9.437	5.37 ± 0.0764	8.29 ± 0.0588	480 ± 0.393	157 ± 0.129
dft-b1024	8.389	4.76 ± 0.0586	7.38 ± 0.0539	425 ± 0.363	158 ± 0.135
fir-b1024	8.389	8.65 ± 0.658	11.1 ± 0.187	758 ± 9.76	752 ± 9.68
dft-b2048	8.389	4.77 ± 0.0583	7.37 ± 0.0530	425 ± 0.322	158 ± 0.120
fir-b2048	8.389	8.21 ± 0.209	10.6 ± 0.140	728 ± 9.58	783 ± 10.3
fir-b4096	8.389	7.61 ± 0.184	9.71 ± 0.113	674 ± 9.18	846 ± 11.5

Table 2: Results from the experiment comparing DFT and FIR separately. This table shows additional information not present in Figure 14.