Coherent De-Dispersion of Radio Pulsar Signals using Dataflow on FPGA

Frank Boerman (frank@fboerman.nl)
Faculty of Electrical Engineering
TU/e, Eindhoven, The Netherlands

***Abstract*—Current state of the art solutions for pulsars search are done on GPU, which have their problems. Using an approach of dataflow programming a design is presented that implements the existing pulsar search algorithms on FGPA. A higher throughput is achieved to such a degree that real time streaming is made possible. Since the dominant part of these algorithms is the FFT, special care is taken to design the FFT parts, most importantly the twiddle generator, fully on chip. A test library is presented to verify the design. Although practical synthesis of the tool used is not yet available, a theoretical analysis is presented on the expected improvement in throughput and hardware cost.**

***Index Terms*—De-Dispersion Astronomy Radio Pulsar Search FPGA FFT Dataflow**

I. Introduction

Radio Pulsars (or Pulsar Stars in popular language) are rapidly rotating heavily magnetized neutron stars. They are invisible to the naked eye, but can be discovered in telescope observations of the universe. Pulsar stars emit a beam of radio transmission, which rotates around its axis in a circular fashion. Pulsars slowly lose rotation speed over time, this energy loss $\dot{E}$ is called the spin down luminosity. The bulk of this energy is lost due to magnetic dipole radiation and pulsar winds. Only a tiny fraction of $\dot{E}$ is actual radio emission, the type of energy that reaches earth and can be observed by us [1].

Pulsar stars emit a beam of this radio transmission, which rotates around its axis in a circular fashion. These emissions are observed on earth as (relative) short radio pulses. These radio pulses can be compared to the effect of the light of a lighthouse near a harbor. Pulsars have very precise timings and are used in astronomy to study phenomena such as relativity theory, gravitational waves and other fundamental physics. For more details on the exact nature of radio pulsars see [1]. The signals of a radio pulsar travel light years through space, or the Inter Stellar Medium (ISM). The ISM consists of cold ionised plasma and can be approximated with known transfer functions from physics theory. The ISM acts on the signal with four effects, of which the dominating one is the effect of frequency dispersion. For a given signal the dispersion can be expressed in a *dispersion measure* or *DM*. Since it is not known a priori, to find the right *DM* to remove this dispersion an intelligent brute force algorithm is needed. The current state of the art, used by for example the radio telescope system LOFAR [2], is moving these kind of algorithms to a parallel execution optimized platform such as the GPU. This so called coherent de-dispersion algorithm, described for GPU in [3], runs many trials in parallel to find the correct *DM* for dedispersion. However GPU's have restrictive hardware structures requiring a specific mapping of the chosen algorithm. Further disadvantages are their large memory overhead and non flexibility. A more efficient and flexible platform would be an FPGA, however these are difficult to design for and verify in a high level manner. In this work a design using a new approach is presented for coherent de-dispersion, using dataflow programming for mapping to FPGA.

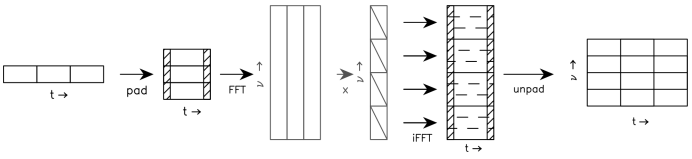


Fig. 1: graphical representation of coherent de-dispersion from [3], showing the splitting of the input into subchannels, converting it to frequency domain and multiplying with the filterbank in several parallel *DM*-trials, then converting back to time domain.

II. Background

A. Frequency (De-)Dispersion

1) The physics models: The theoretical models that the state of the art dedispersion is based on are well defined for some time now in the field of astronomy, an overview of these can be found in appendix A. All references to the *chirp function* mean the transfer function defined in equation 12 in appendix A.

2) DM trials with Coherent Dispersion Measure Algorithm: The Coherent Dispersion Measure trials, or cdmt, is an algorithm implementing the coherent de-dispersion method as a convolving synthetic filterbank, that de-disperses with as high as possible throughput of *DM*'s, originally described in [1] and adapted for GPU in [3]. By splitting the input into independent parts and then utilizing a heavily parallelized computation platform (in the original work a GPU) a high throughput is obtained. The flow diagram is displayed in figure 1.

To describe the algorithm first some parameters are introduced. As introduced before, the observations are split into multiple frequency bands. This is done at the telescope input and lays outside the scope of the algorithm itself. This first split is thus a split in the frequency domain. The algorithm itself splits the timeseries of a single channel into blocks, this second split is thus a split in the time domain. This is an important distinction to keep in mind. In terminology these are referred as a *channel*

and *block* respectively.

In terms of parameters there are n_s input channels, n_c blocks of size N_{bin} samples with n_d samples overlap per block. This means that each block has $N_N = N_{bin} - n_d$ unique samples. These settings are used in n_{DM} trials. Each split block in a timeseries of a channel has n_d samples overlap according to the overlap-save method, to prevent pollution by the wrap around region of the Fourier transform. The amount of overlap n_d for fully preventing pollution is ideally based on the sample rate of the input and an estimate of the dispersion within the channel. However to vastly reduce complexity a fixed n_d will be used. This will reduce the efficiency of the algorithm slightly, but makes the time series blocks fully independent from eachother and thus better parallelizable.

In the original implementation on GPU the parameters are tuned to work with input from the LOFAR telescope system [2], for fair comparison these will be used in the implementation of this work: $N_{bin} = 2^{16}$, $n_d = 2^{11}$. The n_c depends on the total length of the timeseries input N: $n_c = \frac{N}{N_N}$ in which n_c needs to be a positive integer.

The core of the algorithm is as follows, each block is independently de-dispersed with a given *DM* in the frequency domain, by converting to frequency domain, calculating the *chirp*, multiplying with the *chirp* and converting back to the time domain. Then all resulting blocks have their padding removed and are concatenated together to form the output signal. The beauty in this algorithm is that because of the choices of parameters and setup described above, the workload is split into many independent smaller problems, which can then be executed with a high degree of parallelization on a platform. The more formal definition of cdmt is given in algorithm 1 in appendix E.

In a parallel *DM*-trial the *DM* dependent part can be duplicated and executed in parallel. Since the forward Fourier Transform is independent from the *DM*, its result can be reused for multiple *DM*-trials, everything after that will then be duplicated.

To check if the correct *DM* is found some form of candidate selection needs to be done. For example peak detection can be done with filters taken from the medical field in the form of the AMPD algorithm described in [4]. This could be implemented in dataflow using adjusted versions for FPGA, examples include [5]. However candidate selection is out of the scope of this project and will not be discussed further. The focus for simulations will be on single *DM*-trail.

From this description it follows that an important implementation part is the conversion from frequency to time domain. For this the Discrete Fourier Transform can be used, which is introduced shortly in the next section.

B. The Fourier Transform

1) FFT Radix-2: The Discrete Fourier Transform is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}, k = 0, 1, ..., N-1 \quad (1)$$

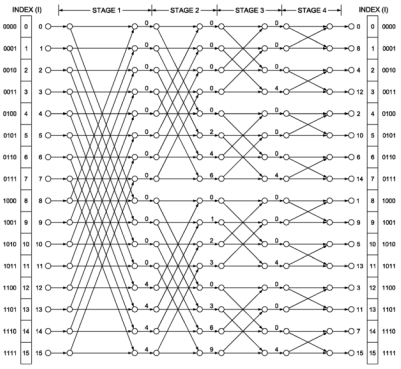


Fig. 2: radix-2^2 FFT flow diagram for a $N = 16$ FFT, taken from [10], the first and last column of numbers represents the indices of the input and output vector of samples. The columns in between represent the rotation factor ϕ

in which

$$W_N^{nk} = W_N^{\phi} = e^{\frac{-j \cdot 2\pi \cdot \phi}{N}} \quad (2)$$

For implementing the DFT of size N, in which N is a power of 2, the *Cooley-Tukey* FFT is used [6]. This *divide and conquer* type algorithm recursively breaks up the DFT into 2 smaller DFT's per iteration of size $N = N_1 N_2$, in which N_1 and N_2 are also a power of two. This reduces the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N log_2 N)$.

The FFT is calculated in a series of so called stages $S = log_r(N)$, the r here is called the *base of the radix*. To decrease non trivial rotations a further optimization is used in the form of radix-2^2, which is detailed in appendix F. The flow diagram for base 2 radix-2^2 is depicted in figure 2 [7]. The pattern of crossing wires in the flow diagram are named butterflies, and each knot in the flow diagram represents a complex rotation ϕ defined in equation 2. These rotation factors are called twiddles. From equation 2 it follows that there is no rotation needed for $\phi = 0$ and for $nk = \phi = [N/4, N/2, 3N/4]$ the rotations are trivial: $W_N^{\phi} = [1, -j, -1, j]$. These latter rotations can be executed trivially by swapping the real and complex components and flipping the sign of the complex part.

A higher radix base r could be chosen, for example radix-4. The number of inputs of each knot in the flow diagram would then be four. One butterfly in radix-4 is then a rewritten double radix-2 butterfly. This results in a reduction of multiplications by 25% and an increase in the number of additions with 50%, as proven in [8] and measured in [9]. From dataflow programming mapping to dsp slices in the FPGA, it follows that for every accumulator there is also a multiplier present. So a trade-off of less multiplications but more additions does not have any actual gain in our programming model. Therefor radix-2 is the chosen radix base.

2) Sample Ordering: DIF & DIT: One of the consequences of the FFT is that the order of samples is changed. In what way depends on the type of FFT: Division in Frequency (DIF) or Division in Time(DIT). The difference between these is the way the splitting of the DFT in half DFT's is chosen. This results in that either the input or the output is

in *bit reversed* order, the other side is then in *natural order*. DIF is *natural order* input and *bit reversed order* output, DIT vice versa. *Bit reversed order* means that the indices of the data are transformed by the reversing of their binary representation. For example with $N = 16$ the index 2 is 0010 in binary, thus in bit reversed order this is then changed to $0100 \rightarrow 4$.

C. FFT Architecture Options

For a FPGA implementation architecture of the FFT a mapping needs to be done from the flow diagram in figure 2 to actual hardware. There are four categories, each a combination of two choices. The first choice is the number of samples taken in per clock cycle, so is the architecture parallel or single. It is essentially a form of loop unrolling, in that multiple butterflies from the flow diagram are mapped to multiple parallel hardware blocks. This structure has higher throughput for lower clock speeds, in exchange for higher hardware cost compared to single. The second choice is if there are feedback loops present. Feedback loops enable reusing hardware blocks for multiple butterflies and rotators, wich are mapped to a single hardware block. They reduce hardware cost for rotators and butterflies in exchange for higher memory usage with their buffers. The four possibilities are then called Single-path Delay Feedback (SDF), Multi-path Delay Feedback (MDF), Single-path Delay Commutator (SDC) and Multi-path Delay Commutator (MDC) architectures class for serial feedback, parallel feedback, serial feed forward and parallel feed forward respectively.

For dedispersion there is a need for high throughput with memory hardware as the limiting factor so memory hardware cost needs to be minimized. Furthermore a high scalability of large FFT size N is required. Many options can be found in literature, showing the mentioned trade offs. SDF with high $N = 1 \cdot 10^6$ but very large buffers in are discussed in [11]. Options for MDF can be found in [9], [12] and [13], also featuring high buffer usage but also N orders of magnitude lower. The number of parallel samples per cycle P is also relatively small with P up to 8. Most SDC designs focus on achieving high speed, but in doing so only have a small (< 1024) possible FFT size N or execute unnecessary (for dedispersion) reorder operations such as in [14].

In [10] a MDC is presented that features highly scalable N and P configuration using simple hardware blocks with low buffer usage. It has the highest possible P as well as efficient sample grouping per parallel path to reduce rotator hardware cost. Thus [10] will be used as the basis for the implementation for the proposed cdmt *DM*-trail design.

D. Garrido Parallel FFT

The architecture described in [10], an example configuration shown in figure 3, has two parameters: N point FFT with P samples in parallel per clock cycle. Both N and P need to be a power of two. The system is divided over $S = \log_2(N)$ stages, corresponding with the FFT flow diagram. The samples are shuffled by either input ordering, wiring between stages

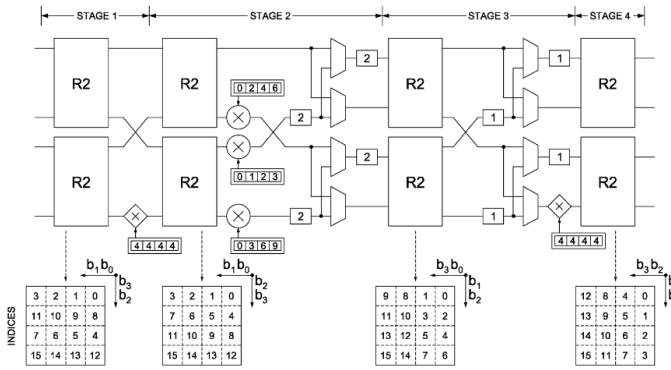


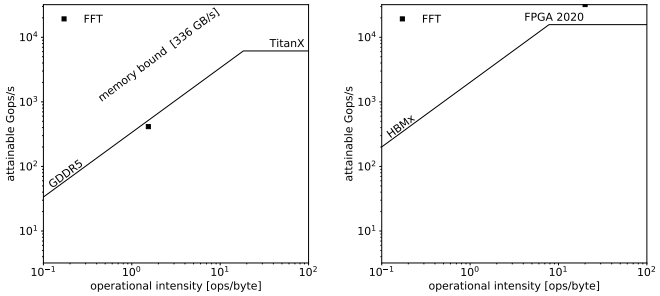
Fig. 3: $P = 4$ $N = 16$ DIF FFT Garrido architecture from [10]. The matrices are the indices of inputs corresponding to figure 2. Samples flow from left to right, so the matrix is read column wise per clock cycle, starting with the right most. Thus each row represents the inputs over time, from right to left for specific input port. Circles denote non trivial rotations, diamond shapes trivial rotations, angles denoted with integer ϕ so $W_N^{\phi} = \exp[-j\frac{2\pi}{N}\phi]$ in the squares next to the rotator.

or shuffler buffer blocks. This is done in such a way that it achieves two goals. First the correct samples are taken together in a butterfly (called *R2* operation due to acting on 2 samples at a time) corresponding to the wiring of the flow graph of figure 2. Second samples are grouped in three categories for rotations: no rotation, trivial rotation, non trivial rotation. This minimizes the number of rotator blocks needed and thus a lower hardware cost in terms of expensive multipliers. For example from figure 3 it can be seen that all samples grouped at the upper most edge have no rotation at all, depicted by the lack of circles or diamonds on those edges.

E. Dataflow programming on FPGA

The implementation method used in this work is dataflow programming, specifically synchronous dataflow. Dataflow as a general concept can be used to model complex systems as independent actors with communication in between. Its basics can be found in [15]. In most literature dataflow is used as an abstract flow concept, however dataflow programming describes dataflow graphs that can be executed with actual data, instead of only abstract flow. In other words when the dataflow graph is executed with actual input data, actual output data (as opposed to abstract observations) is the output. For the implementation part of this project a new tool called StaccatoLab will be used. It allows a designer to create dataflow graphs in a python environment, simulate them and finally map them to FPGA. An theoretical overview of StaccatoLab can be found in [16] and a more practical overview in its documentation in [17]. At the time of writing the FPGA mapping part was not done yet. Therefor only theoretical and simulations results are presented in later sections.

A dataflow graph consists of nodes and edges, with tokens holding data flowing over the edges, nodes consuming and producing them. Each node executes a defined action (called an output function fo) on its input token, producing a output



(a) Roofline for single FFT on GPU

(b) Roofline for single FFT on FGPA

Fig. 4: A roofline plots the operations per byte on the x-axis and the attainable Gops/s on the y-axis. A point on the sloped part of the graph means the operation is memory bound, and thus inefficient in terms of available hardware resources. The straight part is computational bound.

token. A single token can hold multiple values in a tuple. The StaccatoLab system enforces output functions in such a way that it can be analyzed and pipelined by StaccatoLab for FPGA mapping. It is possible te define a Finite State Machine for a node, changing its fo every set number of cycles, this results in a Cyclo Static Dataflow Graph (CSDF). The behavior of the specific type of dataflow graphs used in StaccatoLab is strictly defined, a summary of its rules are detailed in appendix B.

III. Problem Statement

Current implementations of coherent de-dispersion are achieved by a GPU [3], which leads to inefficient hardware usage. Figure 4 plots the so called *roofline* for a single FFT on GPU (4a) and on a typical FPGA (4b). This shows the operational intensity of a single FFT on both platforms, which is the total number of arithmetic operations divided by total amount of memory traffic needed for a procedure. For one FFT the operational intensity is approximately 1.54 on GPU and approximately 20 on FPGA, see appendix H for details on calculating this.

In figure 4a it is shown that the FFT on GPU as used in [3] is memory bound and not computation bound, meaning its performance is limited by availability of memory bandwidth and not computation resource. Thus the usage of computation resource is inefficient. This is due to the structure of the GPU requiring moving data from device to host every three FFT stages as shown in [18].

This roofline shown is only for a single FFT, the cdmt requires many more. Even worse the throughput combined with the hardware structure of needing extra copy passes between device and host described in [3] is also too slow for real-time processing, instead observations are saved to disk and processed later, which requires very large storage capacity. It would be advantageous to redesign the cdmt procedure such that it has higher hardware efficiency and enough throughput to enable the possibility to stream the data through the system while processing it to enable realtime processing and a re-configurable parameterized design.

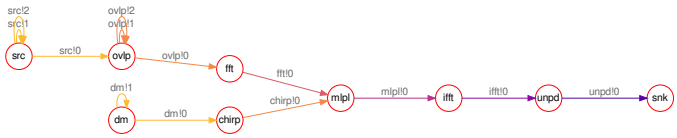


Fig. 5: Dataflow graph depicting high level simulation of single *DM* trial. $ovlp$ executes overlap saving method, $fft, ifft$ depict the forward and backward fourier transform respectively, $mlpl$ depicts elementwise multiplication, $dm \rightarrow chirp$ generates the chirp function for given *DM*, *unpd* removes the overlap saving.

A candidate for these requirements is a FPGA platform. Implementing such a system on a FPGA has however multiple disadvantages compared to a GPU. On a FPGA everything needs to be defined by the designer, there is no implicit parallelization support. By default there is no floating point support, requiring either adding this support or writing the design in fixed point arithmetic and finally the number of different arithmetic operations possible is constrained to addition/subtraction and multiplication. It shares the problem with GPU of being tedious to simulate and verify a possible design, and parameters such as throughput are difficult to predict before actual measurements.

To tackle this problem, dataflow programming in StaccatoLab is used to create a parallel design that is guaranteed to meet throughput requirements that enable real-time processing of telescope data. Using Staccatolab a simulation can be build to tune the throughput of the design before implementation. The output functions of the node are defined such that fixed point arithmetic is used with only elementary arithmetic operations that have native support on FPGA platform. The resulting design will then be simulated and verified. While the properties of the final FPGA hardware platform will be taken into account, the final step of synthesis to this platform from the dataflow graph is not yet ready in StaccatoLab and will thus not be discussed.

IV. High Level System Design

A. Simulation

To do a throughput and memory analysis of cdmt, a high level dataflow implementation is built in StaccatoLab and its graph is plotted in figure 5. High level means here that the output function of each node is a library function that cannot be synthesized, which implements the a large operation such as a FFT. The output result of simulating this data flow graph is depicted in figure 6. To generate a test input signal for this simulation, first a pulsar signal is generated with a C program supplied by the author of [3]. Then a forward dispersion function is calculated for a chosen *DM*, this means applying the dispersion on the signal instead of removing it with dedispersion. In figure 6 this is depicted in green and red. It can be clearly seen how the peaks from the green graph disappear into the noise floor of the signal in red.

Then this signal is split into blocks of size $N = 2^{16}$ and put in the *src* actor. In this high level simulation a single token holds

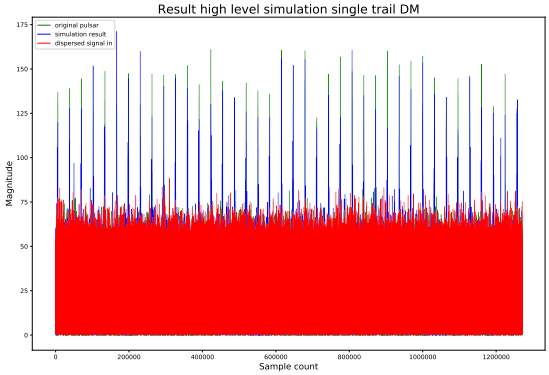


Fig. 6: Output of high level simulation, on x-axis sample count, on y-axis magnitude level, unit is from telescope input and not relevant for discussion

an entire block of samples. Each node does its operation on the whole block at once.

The result of the simulation is shown in blue in figure 6. It can be seen that although these peaks are at the same position as the green original signal, they sometimes vary in magnitude with respect to the original green signal. This is due to the effects of splitting the signal into blocks for de-dispersion.

B. Throughput by Design

The big advantage of data flow programming is that a graph can be designed with an exact throughput of samples per clock cycle as a hard requirement, instead of measuring the implementation after designing and synthesis. The goal for this project in terms of throughput is to enable real time processing of telescope data, compared to the processing after capturing method described in [3]. The dataflow graph will be designed such that it can handle one token per cycle continuously.

To properly compare the resulting design in this work with the original method in [3] the same parameters are chosen in terms of FFT size this means FFT size $N = 2^{16} = 65536$. Telescope data is digitized as 2 bytes per complex sample, one for the real and one for the imaginary part. From the description in [3] the following full setup is defined. The observations are done over 200 dual-polarized sub-bands with a bandwidth of 195.32 kHz each, for a total bandwidth of $2 \cdot 200 \cdot 195.32 \times 10^3 = 78.128 \times 10^6$ samples per second. These samples are saved to disk, to be then processed by 23 nodes, each having 4 NVIDIA TITAN GPU's, over 80 *DM*-trials. Thus for real time processing and with 1 sample per clock cycle of the input bandwidth a design needs to be able to process 78.128 Msamples/s per *DM*-trail.

From the high level design it follows that the dominating part in terms of complexity and memory usage to implement, is the FFT. When looking at a typical synthesis of a FFT on FPGA as described in [19], for a modest FFT parallel implementation with $P = 16$ a clock frequency of 143 MHz can be expected. This would result in a maximum throughput of 2.29 Gsamples/s, which in turn results in the ability to process $\frac{2.29 \times 10^9}{78.128 \times 10^6} = 29$ trails in parallel in a single FPGA board. The situation in [19] is from relatively old hardware, so it can be expected to even go above that.

The output per trail is defined as a single stokes parameter per polarization pair digitized as a single byte. This results in terms of data traffic in an input data stream of 78.128MHz times times 2 bytes per sample plus an output data stream of 39.06MHz times 1 byte per sample per *DM*-trail. For the suggested 29 trails this results in a total of 10.32 Gbit of continuous traffic, which would easily fit with a 100 Gbit ethernet interface that is present on state of the art FPGA boards.

One important remark needs to be made for the StaccatoLab synthesis. For mapping to an efficient pipeline system with minimized register use by the library it is advantageous for the design to put as much computational complexity as possible to be concentrated in the output function fo. This has two reasons. First of all this lets the tool do the job of optimizing the details of the pipe-lining of the computation expression, allowing the designer to focus on the overall design. Second if the design was instead spread over many nodes this would force a lot of long wiring between them, which might not be necessary if wiring logic can be optimized within the node expression.

In order to achieve this a SIMD approach is taken. Instead of a single complex value per token the value of a token flowing through the FFT graph is chosen as a vector of P samples, in which P is a parameter depicting the number of parallel samples per clock cycle. This means that all nodes execute their action on input vector V_P^i producing an output vector V_P^o of equal length P. Then instead of P nodes per stage to execute the butterfly actions there is one node executing P actions. This means that the wiring needed is vastly reduced. There is now a single path from one stage to the next, instead of P wires, satisfying the mentioned routing simplicity. Since the wiring constitutes a shuffle in itself this does raise the need for a sample shuffling within V_P^i in the respective nodes. This approach has the added benefit for the designer of vastly reducing necessary coding complexity for the changing parameter P.

V. Garrido FFT Architecture in Dataflow

A. Overview

As introduced in section II-D the FFT of [10] will be the basis of the design. This subsection will first discuss how the overal architecture is changed to make it work in dataflow, then conversion of all separate building blocks to dataflow is discussed in the subsections below. Apart from redesigning for dataflow all blocks are parameterized with parameters FFT size N and P parallel samples per clock cycle, resulting in a program that can generate a design with design space $N > 0, N \leq 2^{16}, P \in \{4, 8, 16\}$ and type DIF or DIT. This design space is chosen such that the FFT size as in [3] is possible and typical numbers of synthesis are available in [19] for the choice of P.

Besides the SIMD approach mentioned in the previous section two other important steps are taken for the conversion. First the trivial rotators (diamond shapes in figure 3) are merged together with the $R2$ block. This makes it possible to treat

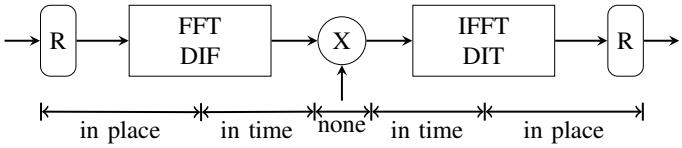


Fig. 7: flow diagram of the fourier transforms, bottom arrows indicate shuffling area of samples

the in and output vector as a single complex operation on the whole token, instead of an operation on only part of the token, resulting in a better synthesizable fo.

Second the meaning of a stage is redefined. The stage number s starts at 0 instead of 1 and each stage is defined as centered around the $R2$ block with optional shuffle block and optional non trivial rotators. This reduces the code generation complexity by having one always present block ($R2$), with blocks after and before it given two simple independent rules: is the stage number odd then rotators are needed, does the stage number satisfy the shuffle requirement (introduced later, depends on type of FFT), then a shuffle block is inserted.

This then results into four building blocks to compose the architecture of: the butterfly block $R2$, non-trivial rotators, shuffle blocks and a twiddle generator block. On top of that there is a specific connection pattern. Both a DIF and a DIT version of the system will be built, the original description in [10] only presents the DIF so this requires a short analysis.

From the FFT flowgraph it follows that DIT is a *transpose* of DIF. Transpose means that the flow diagram is reversed and read from right to left instead of left to right. By reversing all parts it is possible to built a DIT FFT. After analyses it is found that each part except for the connection pattern is a transpose of itself and thus by simply reversing the order of the blocks the whole system can be transposed.

B. Sample Ordering In/Output

Property of a traditional FFT is that it bit reverses the indices of all samples. For a DIF FFT this means natural input, bit reversed output, for DIT FFT vice versa. This is depicted in figure 2 at the right most column. For a hardware design the output order is up to the designer, since wires can be trivially reordered. For the proposed cdmt dataflow system the output is required to be in natural order in the time domain to enable further processing. The orignal FFT architecture from [10] maps the hardware such that the order follows the traditional FFT mapping, thus in bit reversed order. The architecture actually does not execute bit reverse in all samples, but only in samples which differ in the lower $\log_2(N) - \log_2(P)$ bits. I.e. samples with indices differing in the first $\log_2(P)$ bits already need to be reordered before putting into the system. The remaining samples are reordered by the connection pattern and the shuffler blocks. The first only changes samples over input ports (vertical in the scheme), the second also reorders in time (horizontal in the scheme).

For DIT the system is transposed. Then first the *in time* shuffles are done and then only *in place*. By putting them after each other the two shuffling cancels. Thus the full flow of FFT for the de-dispersion is constructed as depicted in figure 7. At the bottom the shuffling area is indicated. The R denotes the procedure of reordering of samples with indices differing in the last or first $\log_2(P)$ bits and the X denotes the multiplying with chirp function in frequency domain. This means that the chirp function output needs to be in bit reversed order. The output of the whole system is however in natural order. By keeping the samples in bit reversed order for the multiplication domain and reshuffling the chirp function there only needs to be one (for the chirp function) bit reverse shuffling performed instead of two (one for FFT, one for IFFT), resulting in higher hardware efficiency than the GPU implementation. The shuffling of R is formalized in algorithm 2 in appendix E. The R itself is implemented as a memory operation, in which a memory block puts the incoming or outgoing samples at the right address. A full circuit for this can be designed too, but falls outside the scope of this project.

C. Connection Pattern

In the original description of [10] the pattern of wires between the stages is not discussed in detail, for the dataflow implementation it was deduced from the flow graphs, its definition given in algorithm 3 in appendix E. Due to the choice of SIMD approach for the dataflow design, the term *connection pattern* is slightly misleading after conversion to dataflow. There is only one actual edge connected between stages, but in the fo of each node the connection pattern is injected, in which the indices for the input vector are translated according to the pattern. So there is a specific shuffling of order of samples within the input token before executing the main operation of a node.

D. R2: Butterfly

The $R2$ butterfly block is the most trivial of all blocks and requires not a lot of adjustment. A radix-r butterfly is defined as having r inputs $I_{0...r}$ and r outputs $O_{0...r}$ which for our radix-2^2 results in the relation:

$$\begin{aligned} O_0 &= I_0 + I_1 \\ O_1 &= I_0 - I_1 \end{aligned} \tag{3}$$

The dataflow adjustment needed here is due to using the SIMD approach split up V_p^i into $\frac{P}{2}$ parts of length 2 and applying the above relation to each subpart, which then concatenated forms the output vector V_p^0.

In addition to this the dataflow graph has the trivial rotation embedded in the $R2$ block. This is done in all odd stages at every third index in blocks of four of the vector, defined in equation 4.

$$(2 \cdot i + 1) \bmod 4 = 3, i \in \{0, ...P\} \tag{4}$$

This trivial rotation is done on the inputs when using DIF and on the outputs when using DIT, this is due to transposing the block.

On top of that on all odd stages except the last one, a vector of twiddle factors is given as an extra input. These are then multiplied element wise, for DIF on the outputs of relation 3 and for DIT on the inputs.

These rules create an output function with large number of

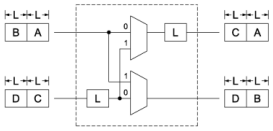


Fig. 8: circuit of data shuffler from [10]

non complex operations. It consist of only three unique trivial operations: addition/subtraction, swapping real and imaginary part (so address translation) for trivial rotation, and the indices translation of the connection pattern which is present in all blocks.

E. Sample Shuffling

The data shuffler essentially transposes an matrix input of $2L \times 2L$ samples, taking the samples as blocks of length L samples. The original circuit design of the shuffler can be seen in figure 8. There are always *num_shuffles*$= \log_2(N) - \log_2(P)$ shuffle blocks in the system, this stays unchanged in dataflow. In dataflow graph a shuffle block is inserted after (for DIF) or before (for DIT) the $R2$ blocks for all odd stages that satisfy $s \geq S - $*num_shuffles* for DIF and $s <$ *num_shuffles* for DIT.
The dataflow block is implemented with 2 self loops to serve as the buffers and a FSM in each node (as defined in a CSDF graph, see subsection II-E) to represent the control circuit of the multiplexors as depicted in figure 8. The FSM has 3 states: initialization, in which the buffers are filled with initial values, state which represents the multiplexor setting of 0 and state which represents the multiplexor setting at 1. These are numbered state $\{S_0, S_1, S_2\}$. The FSM stays L cycles in each state before a transition. In equation 15 in appendix E the states are formalized with inputs I_0 I_1, buffers L_0 L_1 and outputs O_0 O_1 for states S_0, S_1, S_2.
As with the $R2$ block described in section V-D this is an operation on two complex values, so the dataflow node needs to execute the above relation on two complex values samples from V_p^i at a time and concatenate the result, in the same fashion as with the $R2$. This then requires P self loops with a queue capacity of L each to act as the buffers. The output from equation 15 is then simply concatenated in the same way as for the $R2$.

F. Twiddle Generator

1) Architecture Analysis: In the original architecture description the non trivial rotators are implemented as a complex multiplication with a complex number of a corresponding angle called a *twiddle factor* or *twiddle*. This description gives no recommendations about how to generate these twiddle factors. From other literature many ways can be found, however most of them focus on small static sized FFT and put large number of pre-computed twiddle factors into memory, such as in [21] or with additional smart addressing methods to reduce memory usage in [22]. Since one of the focuses of our design is hardware efficiency, a twiddle generator is preferred, which calculates the twiddle factors on runtime and does not depend on large static memory. Existing implementations of these are for example [23] and [24]. Both of these present a method to generate twiddles with variants of the cordic algorithm [25] by refactoring the twiddle formula, however they are for small FFT size only [23] and non parallel hardware [24]. For the dataflow parallel version a new method is presented here that takes advantage of the specific order of twiddle factors needed for the implemented architecture, which is analyzed first.
For analysis the resulting angles from radix-2^2 are rewritten for easier reading. The rotation angles are defined with integer ϕ as $\phi = \frac{\gamma \cdot N}{2\pi}$ with γ as the rotation angle in radians, corresponding to the complex value $e^{-j\gamma}$. This is the same format used in figure 3, depicted in a series of squares next to the non-trivial rotators drawn as circles. Due to the SIMD way of executing the stages, the input under analysis is a rotation vector V_P^R of length P with at each place the rotation angle, which can be zero, per parallel path. Then a matrix M_P^R can be defined by concatenating these for each clock cycle column wise, with the most left column the first in time. For larger N and P a pattern can be deduced, to show this the matrix for $N = 64$, $P = 16$, $s = 1$ is shown here:

0	0	0	0	0	16	8	24	0	8	4	12	0	24	12	36
0	2	1	3	0	18	9	27	0	10	5	15	0	26	13	39
0	4	2	6	0	20	10	30	0	12	6	18	0	28	14	42
0	6	3	9	0	22	11	33	0	14	7	21	0	30	15	45

When looking at the first row it can be seen that it can be split into four groups which are a multiplication of base number, which then is ordered with even indices first. For example the second group of 4 numbers in the first row is $\{0, 16, 8, 24\}$, which are all a multiplication of 8, ordered with indices $\{0, 2, 1, 3\}$. Or in other form $8 \cdot [0, 2, 1, 3]$. When taking these basis for each of the four groups of four angles, you get $\{0, 8, 4, 12\}$ which is the same indices pattern with base number $\lfloor \frac{N}{P} \rfloor = 4$. Then the following row is simply 1 added to the base numbers, or equivalently adding $[0, 2, 1, 3]$ to each subgroup. In other words the whole row can be calculated by an expansion from the third value of the row onward, starting at 0 for the first row. From there the base numbers for all subgroups can be deduced and subgroup expansion applied.
Although this observation is done for this specific setup this not only holds for this example but for all implementations within the design space of $N > 0$, $P \in \{4, 8, 16\}$ with N a power of two, *but only for the first rotations stage* $s = 1$.
For $s > 1$ the pattern is a little different. The first group of four numbers is still done in a similar way but now instead of incrementing the base with 1 it is incremented with 4 to the power of $\left\lfloor \frac{s-1}{2} \right\rfloor$ until this exceeds $N/4$. Finally the other subgroups are simply copies of that first group, with no extra expansion. In other words repeat the first four columns $P/4$ times. Again this holds for the whole design space of $N > 0$, $P \in \{4, 8, 16\}$.
In algorithm 4 in appendix E this pattern is formalized, here $odd(V)$ and $even(V)$ are helper functions that select all odd or all even numbers from the given array respectively. Converting this result into dataflow can be built by defining

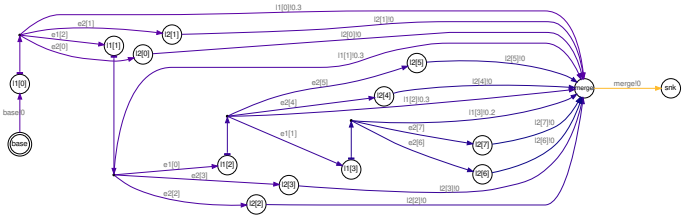


Fig. 9: Dataflow twiddle generator taken from StaccatoLab. Please note that names with an exclamation mark are auto generated edge names and not relevant for discussion.

three blocks: a baseangle α generator using the CORDIC algorithm introduced in section V-F2 called $base_\alpha$, a rotator with a fixed rotation of $\phi = \frac{N}{P}$ called $R_{N//P}$ and a rotator which rotates with a integer multiple of the angle of the input called $R_{n\alpha}$ (with $n \in \{2,3\}$). Integrating this together is shown in figure 9 for $s = 1$ and $P = 16$, here l1[i] are the $R_{N//P}$, l2[i] the $R_{n\alpha}$ and *base* is the base angle generator. Per clock cycle there is always one base angle, which is then expanded $P/4$ times with $R_{N//P}$ static rotators. Each of these is then expanded four times with the $R_{n\alpha}, n \in \{2,3\}$ rotators finally being merges into one vector of P twiddle factors, inserting zero rotations at every fourth place. Each of the rotators consist of one 2D matrix multiplication with an incoming complex value, which in turn consists of a tuple of 2 integers representing the real and imaginary part.

2) Base Angle generator: The above method for twiddle generation requires a base angle, which can be generated by the CORDIC algorithm (also known as Volder's algorithm). It approximates rotating a vector with arbitrary angle α. The implementation of the CORDIC is dominant in the error of the twiddle factors, which in turn is the dominant in the overall error of the whole FFT. Its choices are therefor important for the final result. For this design the basic form rotation mode CORDIC is used, since it can rotate a vector with a desired arbitrary angle. This is used to rotate the standard vector $(1,0)$ with a desired base angle, which can then be further expanded as discussed. It is the most basic form of CORDIC, which is selected to keep down the complexity of the project due to time constraint. The algorithm is expressed as a series of state equations per iteration in equation 14 in appendix D taken from [25]. These are implemented as a cascade of dataflow nodes that execute these state equations, with tokens holding a vector of all states. In the final output token $i = l$ with the final state the desired base angle for twiddle generation is then the tuple (x_l, y_l) which depicts a coordinate on the complex plane corresponding to the desired angle.

3) Static Rotators: To construct the output function fo of the two rotator type nodes, the rotation matrix is rewritten. For an angle represented as a complex value with real part x and imaginary part y and a rotation angle α a rotation is defined as shown in equation 5.

$$R(\alpha) \cdot V_C = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{5}$$

$R_{N//P}$ can then be represented as a single matrix multiplication with precomputed elements and thus in 4 multiplications and two additions for the output function. For $R_{n\alpha}$ the input already has an angle, and that angle needs to be multiplied by 2 or 3. The input can be seen as a rotation with the base vector $[1,0]$ with angle α. So the desired output angle $n\alpha, n \in \{2,3\}$ can be seen as rotation of that same base vector but with the desired angle. For a complex value it holds that: $x + yj = \cos(\alpha_0) + \sin(\alpha_0) \cdot j$. Using this and following the trigonometry identities of cos and sin this results in the representations of the output tuple $(x', y') \rightarrow x' + y'j$ for 2α and 3α shown in equation 6, which can be written directly as an output function since it only holds multiplications and additions.

$$\begin{aligned} \begin{bmatrix} x'_{2\alpha} \\ y'_{2\alpha} \end{bmatrix} &= \begin{bmatrix} \cos(2\alpha_0) & -\sin(2\alpha_0) \\ \sin(2\alpha_0) & \cos(2\alpha_0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x^2 - y^2 \\ 2 \cdot x \cdot y \end{bmatrix} \\ \begin{bmatrix} x'_{3\alpha} \\ y'_{3\alpha} \end{bmatrix} &= \begin{bmatrix} \cos(3\alpha_0) & -\sin(3\alpha_0) \\ \sin(3\alpha_0) & \cos(3\alpha_0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \cdot x^3 - 3 \cdot x \\ 3 \cdot y - 4 \cdot y^3 \end{bmatrix} \end{aligned} \tag{6}$$

G. Integration

With all these subparts defined, the final system can be built straightforward. Given are a N and P with N a power of two and $P \in \{4,8,16\}$. The system will consist of $S = \log_2(N)$ stages sequentially with stage number $s \in \{0,1,...S-1\}$. For each odd stage (except for the last one so when $s = S-1$) a twiddle generator is added and fed into the $R2$ block, as well as activating the non trivial rotation in the $R2$. For DIF a shuffle block is inserted before the $R2$ blocks which satisfy $s >= \log_2(N) - \log_2(P)$, for DIT a shuffle block is inserted after all $R2$ blocks that satisfy $s < \log_2(N) - \log_2(P)$. The R transform described in subsection V-B is added as a RAM node which reads or write the data with the correct address pattern.

The IFFT can be expressed in the form of the FFT by swapping the real and complex parts at the in and output of the system. When the IFFT is built an extra is node is therefor inserted before and after the generated FFT system. An example of a DIF FFT $N = 256, P = 16$ system is shown in figure 10. Its complementary DIT IFFT is shown in figure 11.

H. Fixed Point Arithmetic

Because FPGA has no native support for floating point operations the design is based on fixed point arithmetic, contrary to the GPU based floating point original implementation of cdmt. For the dataflow design a complex value is represented as a tuple of two signed integers depicting the real and imaginary part respectively. The original input setup of telescope data is used, so an original input stream of complex values represented by 2 unsigned bytes. Thus the range of both real and imaginary part is $\{0,1,...,255\}$.

1) Overview: The system is implemented using fixed point arithmetic. Each complex value is represented as a pair of two integers, and all operations are defined on this pair. Some choices have to be made in terms of scaling of floating point to fixed point as well as scaling calculations. For this one uniform parameter is defined: Z, which corresponds to the maximum number of bits used in data transfer in the system. There are multiple points at which scaling is needed: scaling the input

Fig. 10: FFT DIF dataflow graph implementation for $N = 256, P = 16$

Fig. 11: IFFT DIT dataflow graph implementation for $N = 256, P = 16$

values to take the full range of available input bits, scaling the FFT and IFFT stages according to the DFT formula, scaling rotation multiplications to prevent overflow and scaling the output values back to match the scale of original input values.

2) Characteristics: First lets look at the data and platform characteristic. As described in [3] the input telescope data is 2 unsigned bytes per complex value. The dataflow graph is mapped from dataflow to FPGA dsp slices. When looking at a typical example of such a dsp slice, depicted in figure 19 in appendix G taken from documentation [26], it can be seen that the maximum bus width for data transfer between two accumulators is 48 bits. This is the absolute maximum that can be mapped to it for accumulator to accumulator data transfer. Most other operations have less bit space, the specifics are up to the actual synthesis process, which wont be discussed in this work. The simulation enforces this maximum at all stages. However for the scaling it is tried to minimize the number of bits needed, captured in the Z parameter, while still retaining high precision. To restrain the scope of this project the most simple scaling methods are used. This means the input is scaled to the full Z number of bits and all subsequent scaling after arithmetic operations are done in such a way that the result is always scaled back into the range of Z number of bits. So the original input values should be scaled up by multiplying all values with 2^{Z-8} and output values correspondingly divided by 2^{Z-8}. The telescope generating input data linear shifts it to be unsigned. This causes a very large DC component in the frequency domain, which thus lowers the accuracy since all bins need to fit in the chosen fixed point of the FFT. To help alleviate this the telescope output is shifted back to signed before the above discussed upscaling to Z bits is performed. Please note that all these steps is done as a pre/post processing so not in the FPGA design. The scaled samples with the chosen Z are then used as input to the simulated FPGA system.

3) DFT Scaling: Scaling within the DFT is defined as $\frac{1}{N}$ times the inverse DFT. In other words the forward FFT is not scaled but the backward is scaled down by factor N. To reduce max bit usage this scaling can be divided over both FFT and IFFT using the *orthogonal method*. This corresponds to scaling both with $\frac{1}{\sqrt{N}}$ which is equivalent in the end result. In the fixed point implementation this means that since N is always an integer and a power of two with $N = 2^S$ this corresponds to $S/2$ total bit shifting for both FFT and IFFT. For even S this means 1 bit every second stage. For odd S the final output scaling needs to be one extra bit. In figure 12 the max number

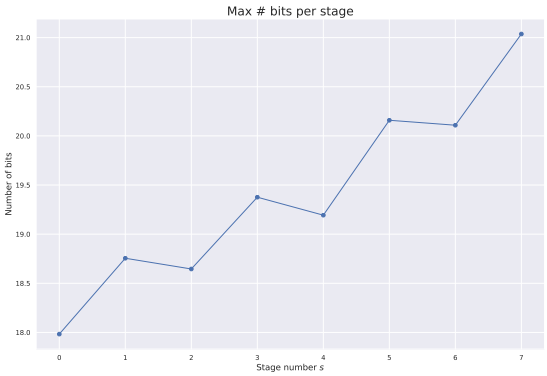


Fig. 12: max bits of input of each stage for DIF FFT with $N = 256, P = 16, Z = 18$, taken from simulation by taking the max values of all inputs seen at each stage

of bits used at the input of each stage is plotted with arbitrary double 8 bit input values, $N = 256, P = 16, Z = 18$ and the explained scaling strategy. The 1 bit per 2 stage scaling can be clearly seen at the input of the even stages, since the scaling is done at the output of the odd stage in the implementation. Please note that this scaling is independent of type DIF or DIT.

4) Multiplication Scaling: In fixed point arithmetic it holds that when multiplying two integers the number of bits needed to store its result is at most twice the number of original input bits, in our case $2 \cdot Z$. The most trivial solution to prevent overflowing is immediately right bit shifting the result with Z. Although there exist other solutions with dynamic scaling per stage, they require more complexity and are out of the scope of this project.

5) Twiddle Generation Scaling: Besides the standard CORDIC scaling, there is another important scaling point in the twiddle generator: the static rotators. As explained in section V-F3 there are two static rotators: $2\alpha, 3\alpha$. The formula for the former is quadratic and needs at most $2Z$ bits before scaling as described above. However the latter has a higher max power of 3 instead of 2. This severely limits the range of Z, since at least $3Z$ bits are needed when the full result is kept. Instead the intermediate product will be scaled. This results in rotation with 2α, scaling down the result, rotation of 1α. Its resulting expression is denoted in equation 7.

$$\left(\begin{bmatrix} x & -y \\ y & x \end{bmatrix} \cdot \begin{bmatrix} (x^2 - y^2) \gg Z \\ (2 \cdot x \cdot y) \gg Z \end{bmatrix} \right) \gg Z \tag{7}$$

VI. FFT Verification

A. Setup and Measure

To verify the implemented FFT, simulations are run in StaccatoLab with generated dataflow graphs with multiple valid configurations of N, P, Z. To verify the correctness of the FFT a test dataflow graph is built with a flow of $src \rightarrow FFT \rightarrow IFFT \rightarrow sink$. Then two checks are done. First the output of the FFT is compared to the output of a standard numerical double floating point precision fft implementation. Second the input of the *snk* is compared to the output of the same FFT→IFFT procedure from the standard implementation. A test is taken as succeeded if these two checks are within a predefined threshold. To compare signals the Quantized Signal to Noise Ration (SQNR) is calculated as a measure of accuracy as defined in equation 13 in appendix C. For comparison a state of the art high accuracy big FFT is selected from [11] which reports a SQNR of 95.6dB for $N = 1 \cdot 10^6$.
The name of the SQNR is misleading, since there are more kind of errors then only quantization error in the system which this formula takes together. However the state of the art FFT work for FPGA uses the SQNR so for fair comparison this is done in this work too.

B. Test Cases

The input signals for the tests have to be chosen in such a way that they can verify the main properties of the FFT and detect the most common implementation problems. The test vectors are based on the findings in [27] which is the basis of the benchmarking method of FFTW described in [28]. FFTW is the most used and cited FFT benchmark and a de facto standard. For all tests it holds that they succeed if their checks satisfy an accuracy approaching the state of the art in [11] of 96 dB, arbitrarily chosen at SQNR of 80 dB.
1) Unit Impulse: Applying a Kronecker delta function with max value of 2^Z will test against overflows as well as the routing of the butterflies without the rotators, because no rotations are needed.
2) Linearity: The DFT is part of the family of *linear transforms*, which means that:

$$\mathrm{FFT}(a_1 x_1[n] + a_2 x_2[n]) = a_1 \mathrm{FFT}(x_1[n]) + a_2 \mathrm{FFT}(x_2[n])$$

must hold for all values of $a_1, a_2, x_1[n], x_2[n]$. So as test input two random noise vectors V_A, V_B are generated and checked if

$$\mathrm{FFT}(V_A + V_B) = \mathrm{FFT}(V_A) + \mathrm{FFT}(V_B)$$

holds.
3) Time Shift: For a linear shift of D samples backwards in time, with wraparound of the samples at the end of the signal to the start, for all DFT it holds that

$$\mathrm{FFT}(V_I^D) = \left[e^{(\frac{-2j\pi D}{N} n)} \right] \cdot \mathrm{FFT}(V_I) \qquad (8)$$

here V_I is the original random signal and V_I^D is the D samples shifted signal. n is the vector $0, 1, ...N-1$. A white noise test vector is used, testing equality between its original and shifted version.
4) Alternative +1 -1: To test the correct butterfly relation, an input vector is generated which continuously steps from the maximum input to the minimum input, in our case between 0 and 255 for an 8 bit input. Please note that this is telescope input so before the additional preprocessing of the system.
5) Random Noise: The last test is white noise input. This is done to check for general correctness of the signal.

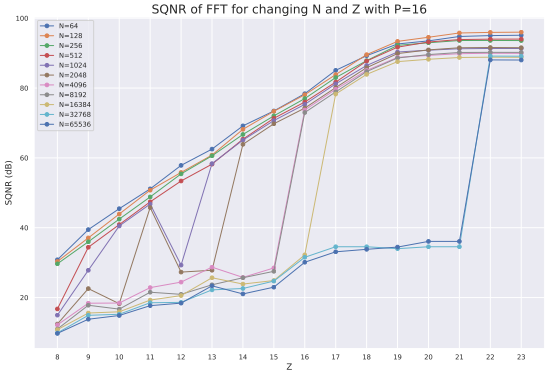


Fig. 13: SQNR of FFT fixed point for changing FFT size N and bit usage Z with $P = 16$

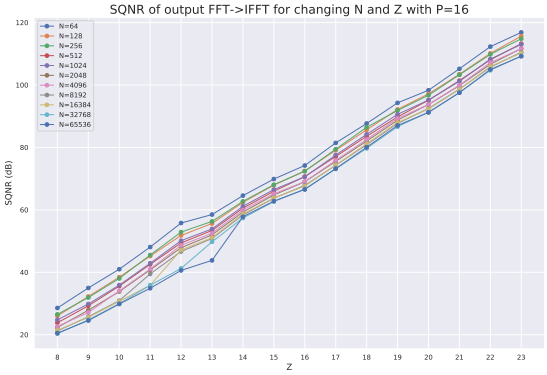


Fig. 14: SQNR of FFT− >IFFT fixed point for changing FFT size N and bit usage Z with $P = 16$

VII. Results

A. FFT Graph

Simulations were run on the graph displayed on a test graph with setup $src \rightarrow FFT \rightarrow IFFT \rightarrow sink$ with parameter range of $N \in \{2^6, 2^7, ..., 2^{16}\}$, $P = 16$ and $Z \in \{8, 9, ..., 23\}$ with as input random white noise to investigate the behavior of the FFT. The results of this are shown in figure 13 and 14. Figure 13 displays the output of only the FFT node, in figure 14 the output of the FFT→IFFT flow is compared to library output with the error is expressed in dB as the SQNR, as described in section VI-A.
Two observations can be made from the result graph: first the accuracy goes up when more bits are used, this is to be expected from fixed point arithmetic point of view, if this was not the case it would have shown an error in the implementation. Secondly for the FFT there is a clear cutoff point in terms of

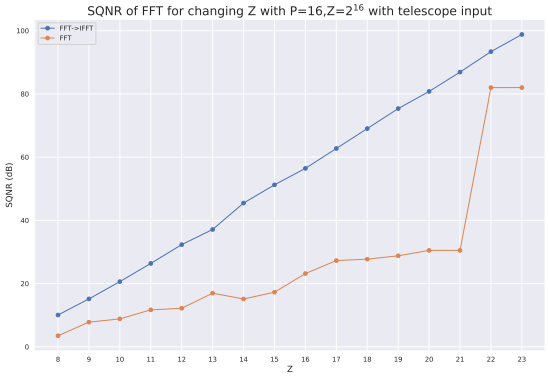


Fig. 15: SQNR of FFT− >IFFT fixed point with telescope input for changing bit usage Z with $P = 16, N = 2^{16}$

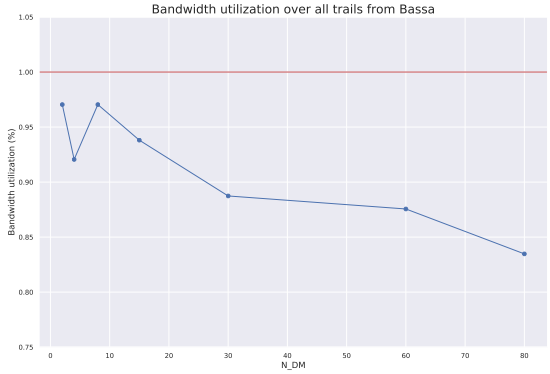


Fig. 16: Bandwidth utilization for benchmark figure 6 in [3] on single TITAN X

Z from which the accuracy improves with a large factor, with this point being higher Z for higher N. This is a interesting result, possibly due to the fact that for a higher N the twiddle factors become very small, thus falling in the margin of error of the fixed point representation. It is not possible to say this with certainty from the presented simulations, further research on this is required. This behaviour is not seen at the output of the IFFT shown in figure 14. This is probably due to the fact that the error made in the FFT in the twiddle rotations has an equal inverse error in the IFFT, thus canceling each other. In the latest FFT on FPGA literature in [11] a SQNR of 96 dB is achieved for $N = 1 \cdot 10^6$. For the desired $N = 65536$ our system achieves 88 dB after achieving the mentioned minimum Z point, in this case $Z = 22$. The literature does not mention any numbers for other N, this remains for further research. This does show that the choices made for the fixed point arithmetic, even though they are relatively simple compared to state of the art solutions, already give an accuracy approaching this state of the art but there is still quite some room for improvement.

To verify the system also holds specifically for the telescope input with its DC component the simulation is run with full block size of $N = 2^{16}$ and $P = 16$ with a sample telescope input, its results are show in figure 15. It shows the same characteristics discussed above. The maximal SQNR achieved here is slightly lower at 82 dB versus the 88 dB, probably due to its large DC component.

B. Comparison cdmt

In the original description of cdmt a benchmark is performed, displayed in figure 6 in [3] (also shown in appendix I). Here a 20 min observation of 10 dual-polarized bands of 195.32kHz bandwidth each with 1 complex sample per clock cycle is used as input, in which a complex sample consists of 2 bytes. The output is given as a single byte Stokes parameter per polarization pair. The cdmt is then performed on a single Titan X GPU. The roofline analysis from section III shows that an FFT on GPU is memory bandwidth limited. This can be further shown by estimating the bandwidth usage of the cdmt results in [3]. From the description it follows that there is 2.2GB output data of one stokes parameter of one byte per polarization pair, so $2.2 \cdot 10^9$ pairs output. Per output pair there are #DM trails and a cdmt pipeline consists of a single shared FFT piped into one IFFT per DM trail. So there are #DM+1 fourier transforms which results in $(B_{max} \cdot T_{proc})/(2.2 \cdot (N_{DM} + 1))$ DRAM traffic available per polarization pair, with $B_{max} = 336.5GB/s$ maximum DRAM traffic [29]. From [18] it follows that the GPU does a full read and writeback approximately every 3 stages of the FFT, with all intermediate results in 32 bit i.e. 4 bytes. This then results in 2 polarization times 2 bytes input, one byte stokes parameter output, and $2 * \lceil 16/3 \rceil$ read and write operations of 2 polarizations times 2 times (complex) times 4 bytes over a single FFT, for a total of 197 bytes per polarization pair per FFT. The used bytes divided by available bytes per polarization pair results in a utilization percentage which is plotted in figure 16. With a minimum of 80% bandwidth utilization this confirms the memory limited scenario. For FPGA there is no extra write backs, so per polarization pair there is only 2 polarization times 2 bytes input and one byte stokes parameter output for a total of 5 bytes DRAM traffic per polarization pair. This finally results in an expected speed up of a little under 40 times. Please note that this is all *without disk IO* since it is very hard to estimate off board memory connections for the FPGA at this stage.

As a power usage estimate the typical max power draw of both boards are taken from the specifications of the manufacturers in [30] for GPU and [29] for FPGA. This gives 250-300 W for the GPU and 225W for the FPGA under typical load. So a small saving in power usage is also expected.

The possibility to stream the algorithm real time (as shown in subsection IV-B) eliminates the need for large disk storage, lowering the hardware cost. As shown in the same subsection IV-B the number of DM trails that can be run on a single FGPA board is an estimated 29 trails. This was done in terms of throughput and memory traffic capacities. Now from the designed dataflow graph an estimation can be made if this also fits in terms of BRAM usage. From the dataflow graph all edges and buffers can be counted to provide estimated number of this BRAM usage. This is heavily dominated (80%) by the buffers (implemented as self loops) from the data shuffles. When taking the $Z = 22$ found in section VII-A

this results in around 222kb for the FFT. Since this does not take into account possible reuse of edges by the library, this is an upper bound. Although the chirp generator and *DM*-trail control circuits take up some memory too, this is expected to be much less since this is comparable to the chirp generator graph, which is only a small part of the FFT. Modern FPGA's boards can have up to 133Mb BRAM on board, thus providing ample space.

VIII. Conclusion and Future Research

The current state of the art solution on GPU for the coherent de-dispersion of radio pulsar signals is severely memory bandwidth limited. By redesigning the algorithm to a dataflow graph on FPGA this limitation is eliminated and a speed up of 40 times can be expected. Real time processing with on chip streaming on the FPGA is possible with this design, removing the need for large data storage for intermediate results. In contrast to the rigid GPU implementation the proposed architecture can be generated dynamically with given parameters of N FFT size, P parallel samples and Z fixed point precision. The testing library created allows the design to be easily verified with these various parameters by making use of the dataflow programming model through Staccatolab.
However the speed and hardware cost improvements are based in theory and estimates following simulations. They rely on assumptions of the underlying StaccatoLab library. Future work is needed to bring these results into practise and investigate wether these assumptions hold in practice. Most importantly that the synthesis process of StaccatoLab is sufficient to not wipe out the gains made possible by its programming model. The dominating part of the cdmt is a first proof of concept. More specific optimizations could be possible. For example the handling of the smallest twiddle factors that possibly fall into the margin of error need to be investigated and improved. Techniques such as the nano rotator from [11] and [31] could help with improving this. Other future work would be the chirp generator. Due to limiting the scope of the project, the chirp generator is not yet fully implemented in a streaming fashion. Clues can be taken from the way the twiddle generator is done, as it has many similarities.

References

[1] D. Lorimer and M. Kramer, "Handbook of pulsar astronomy," 2004.
[2] van Haarlem et al, "LOFAR: The LOw-Frequency ARray," *aap*, vol. 556, p. A2, Aug 2013.
[3] C. G. Bassa, Z. Pleunis, and J. W. T. Hessels, "Enabling pulsar and fast transient searches using coherent dedispersion," 2016.
[4] F. Scholkmann, J. Boss, and M. Wolf, "An efficient algorithm for automatic peak detection in noisy periodic and quasi-periodic signals.," *Algorithms*, vol. 5, no. 4, pp. 588–603, 2012.
[5] A. M. Colak, Y. Shibata, and F. Kurokawa, "Fpga implementation of the automatic multiscale based peak detection for real-time signal analysis on renewable energy systems," in *2016 IEEE International Conference on Renewable Energy Research and Applications (ICRERA)*, pp. 379–384, Nov 2016.
[6] 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.
[7] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing*. Prentice-hall Englewood Cliffs, second ed., 1999.
[8] S. Dirlik, "A comparison of fft processor designs," December 2013.
[9] S. Mookherjee, L. DeBrunner, and V. DeBrunner, "A high throughput and low power radix-4 fft architecture," in *2014 48th Asilomar Conference on Signals, Systems and Computers*, pp. 1266–1270, Nov 2014.
[10] M. Garrido, J. Grajal, M. A. Sanchez, and O. Gustafsson, "Pipelined radix-2^k feedforward fft architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 23–32, Jan 2013.
[11] H. Kanders, T. Mellqvist, M. Garrido, K. Palmkvist, and O. Gustafsson, "A 1 million-point fft on a single fpga," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, pp. 3863–3873, Oct 2019.
[12] Hang Liu and Hanho Lee, "A high performance four-parallel 128/64-point radix-24 fft/ifft processor for mimo-ofdm systems," in *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, pp. 834–837, Nov 2008.
[13] J. Wang, C. Xiong, K. Zhang, and J. Wei, "A mixed-decimation mdf architecture for radix- 2^k parallel fft," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 67–78, Jan 2016.
[14] Z. Wang, X. Liu, B. He, and F. Yu, "A combined sdc-sdf architecture for normal i/o pipelined radix-2 fft," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, pp. 973–977, May 2015.
[15] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, Sep. 1987.
[16] K. van Berkel, "Staccatolab: A programming and execution model for large-scale dataflow computing," in *Multi-Processor System-on-Chip: Vol. 2 – Applications* (L. Andrade and F. Rousseau, eds.), ISTE Science Publishing Ltd, 2020.
[17] C. H. van Berkel, "Staccatolab docs," to be published.
[18] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–12, 2008.
[19] M. Garrido, M. Acevedo, A. Ehliar, and O. Gustafsson, "Challenging the limits of fft performance on fpgas (invited paper)," in *2014 International Symposium on Integrated Circuits (ISIC)*, pp. 172–175, Dec 2014.
[20] Xilinx, *Ultrascale FPGA Product Tables and Product Selection Guide*, 2016.
[21] L. Hongxia and H. Shitan, "High performance algorithm for twiddle factor of variable-size fft processor and its implementation," in *2012 International Conference on Industrial Control and Electronics Engineering*, pp. 1078–1081, Aug 2012.
[22] I. A. Qureshi and F. Qureshi, "Generalized twiddle factor index mapping of radix-2 fast fourier transform algorithm," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pp. 381–384, June 2014.
[23] J. Zhou, "A new method to generate twiddle factor using cordic based radix-4 fft butterfly," in *2013 International Conference on Communications, Circuits and Systems (ICCCAS)*, vol. 2, pp. 505–508, Nov 2013.
[24] T. Hoang, D. Le, and C. Pham, "Vlsi design of floating-point twiddle factor using adaptive cordic on various iteration limitations," in *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 225–232, Sep. 2018.
[25] P. K. Meher, J. Valls, T. Juang, K. Sridharan, and K. Maharatna, "50 years of cordic: Algorithms, architectures, and applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, pp. 1893–1907, Sep. 2009.
[26] Xilinx, *UltraScale Architecture DSP Slice UG579*, september 2019. v1.9.
[27] F. Ergün, "Testing multivariate linear functions: Overcoming the generator bottleneck," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '95, (New York, NY, USA), p. 407–416, Association for Computing Machinery, 1995.
[28] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".
[29] Xilinx, *Alveo U200 and U250 Data CenterAccelerator Cards Data Sheet DS962*, december 2019. v1.2.1.
[30] NVIDIA Geforce, *GeForce GTX TITAN X specifications*.
[31] M. Garrido, K. Möller, and M. Kumm, "World's fastest fft architectures: Breaking the barrier of 100 gs/s," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, pp. 1507–1516, April 2019.
[32] J. B. Observatory, "Frontiers of modern astronomy 4.1."
[33] L. Levin, W. Armour, C. Baffa, E. Barr, S. Cooper, R. Eatough, A. Ensor, E. Giani, A. Karastergiou, R. Karuppusamy, and et al., "Pulsar searches with the ska," *Proceedings of the International Astronomical Union*, vol. 13, p. 171–174, Sep 2017.

APPENDIX A
FREQUENCY DE-DISPERSION PHYSICS

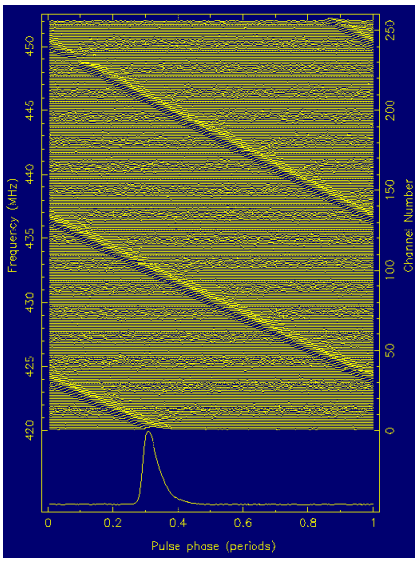


Fig. 17: Pulsar signal observation showing frequency dispersion, taken from [32]

Frequency dispersion is the effect in which signals with different wavelength have different propagation speeds through a non-vacuum medium. When a signal composed of multiple frequencies (and thus wavelengths) is transmitted through this medium, the different frequencies *within* the signal will slowly shift in time from each other.

In figure 17 this effect can be seen from an observation of a radio pulsar pulse signal. Here repetitions of the observed pulse can be seen smeared out over the frequency spectrum in time. The arrival of different frequencies (the vertical axis) at different points in time (the horizontal axis, with as unit the pulsar period) is clearly visible. The result of this is that when looking at the magnitude of the received signal at a telescope the pulse has been smeared out so much that it disappears into the noise floor.

Pulsars signals travel through the ISM. It is not uniform, different paths from a pulsar source to an observation point do not encounter the same effects. This is due to the different column density of free electrons along the traveled path by the signal. To model this effect the Dispersion Measure(DM) is introduced, defined as follows:

$$DM = \int_0^d \eta_e(l)dl \tag{9}$$

Here $\eta_e(l)$ is the amount of electrons at a point along the path, d the distance of the path and l the path itself. This DM is the parameter that will be used in the model of Frequency Dispersion in the ISM. It is not known a priori to an observation of a radio pulsar.

There are multiple ways to model the Frequency Dispersion in the ISM, the two main ones will be presented here. These are tied to the two main algorithms of removing the dispersion, so called de-dispersion. Further details on these models can be found in [1]. These models make use of so called frequency channels. Frequency channels are small bands of equal size in the observed bandwidth. For example one of the largest projects in this field, the SKA, splits a bandwidth of 300 MHz into 4096 frequency channels for the SKA1-Mid system for pulsar search [33].

In the first model, the time delay of the arriving frequency channel for any channel with $f_1 < f < f_2$ is described as:

$$\Delta t = 4.15 \times 10^{-6} \times DM \times (f_1^{-2} - f_2^{-2}) \tag{10}$$

By applying a time shift to the whole channels time series, the dispersion *between* the channels can be successfully removed. This incoherent de-dispersion has the main advantage of relatively low computational cost, due to it only involving a simple shift operation with a constant in the time domain. The disadvantage is that it only negates the dispersion *between* the different channels. There is still dispersion left within the channel. Especially for low frequencies and/or high DM's this dispersion can still be very significant.

The second model describes the dispersion as a phase only filter in the following form in the frequency domain:

$$V(f_0 + f) = V_{int}(f_0 + f) \times H(f_0 + f) \tag{11}$$

here $V(f)$ and $V_{int}(f_0 + f)$ are the observed and emitted signals around a center f_0 and the filter transfer function $H(f)$. This transfer function is then of the following form:

$$H(f + f_0) = exp\left[\frac{2\pi \cdot i \cdot f^2 \cdot k_{DM} \cdot DM}{f_0^2(f + f_0)}\right] \tag{12}$$

here f_0 is the center frequency of a frequency channel and f is the frequency offset within the channel. DM is the dispersion measure as discussed earlier and k_{DM} is the measured constant of proportionality of the ISM model.

By taking the inverse of the transfer function and convolving it with the frequency channel in the time domain the frequency dispersion can be fully removed. Since this is a phase only filter this convolution needs to be done on the complex input signal, so before the traditional squaring of the signal done by telescope systems, due to needing the phase information. So all inputs are complex values. Using the convolution theorem this convolution can be executed by element wise multiplication in the frequency domain, so then de-dispersion becomes the following steps. Convert the input signal to frequency domain, calculate the inverse transfer function for given DM, multiply in frequency domain, convert the result back to the time domain. The $H(f + f_0)$ is usually combined with a function with anti-aliasing properties called a *taper*. This combined function is called the *Chirp Function*.

The advantage of this de-dispersion method is that the dispersion can be *fully* removed, even within the channels. The disadvantage is the high intensity both in computation, due to need to switch to and from the frequency domain, and in data due to needing the full complex signal.

Please note that these model equations follow the physics convention in terms of writing. Again for more details behind

these models see [1].

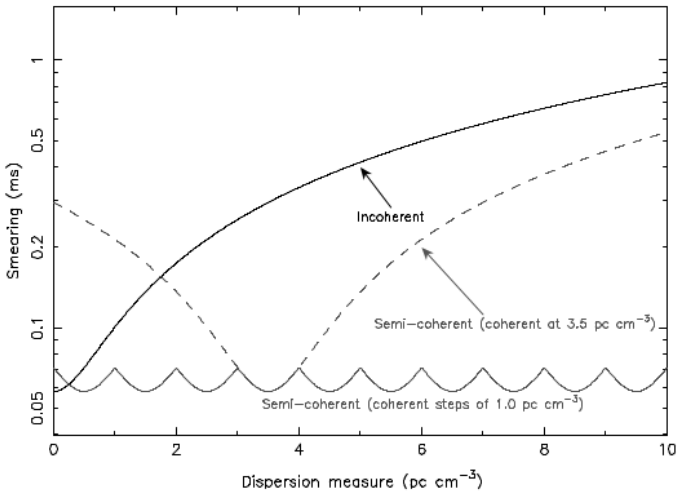


Fig. 18: smearing when using semi-coherent approach, taken from [3]

These two models of de-dispersion can be combined. By using coarse grained trails of incoherent de-dispersion and then switching to fine grained de-dispersion when a pulse seems emerging, the drawbacks of the two models can be negated. This is called semi-coherent approach, as described in [3]. Figure 18 shows the smearing due to the combined effects of dispersion within a channel after using semi-coherent, with finite time resolution, and finite DM steps over the full bandwidth, as a function of DM.

Appendix B
Synchronous Dataflow

For a basic understanding of the presented dataflow graphs the following rules summarize the specific model of synchronous dataflow used in StaccatoLab: * A dataflow graph consists of nodes, that execute actual actions on the data, edges that connect the nodes, in the form of queues, and tokens that flow through the graph according to its rules, consisting of certain data. Execution of a node is also called *firing*.

* Each node is *self-timed*. This means that a node can fire immediately when there are sufficient tokens on each of its input edges

* A graph is clocked, so the firing of firable nodes is synced globally. In practise this means that time is an integer and that a firing can only occur on a multiple of an integer.

* Only one output token per firing: at most one token is produced per output by a firing node

* The queues in edge have a max token capacity *slack*, a node will only fire if there is either enough slack on its output edges, or it is guaranteed that by simultaneous firing of connected nodes enough slack will be freed up

Appendix C
SQNR

$$SQNR(dB) = 10 \cdot \log_{10}\left(\frac{E\{|\, X_{ID}\,|^2\}}{E\{|\, X_Q - X_{ID}\,|^2\}}\right) \tag{13}$$

Here E is the expected value (or mean), X_Q is the output of the operation and X_{ID} is the output of the reference for that operation.

Appendix D
CORDIC states

CORDIC rotation mode state equations taken from [25].

$$\begin{aligned} x_{i+1} &= x_i - d_i \cdot 2^{-i} \cdot y_i \\ y_{i+1} &= y_i + d_i \cdot 2^{-i} \cdot x_i \\ \omega_{i+1} &= \omega_i - d_i \cdot \tanh(2^{-i}) \\ d_i &= -sign(\omega_i) \end{aligned} \tag{14}$$

Here the states are described as: x_i and y_i for the real and imaginary part of a vector on the complex plane representing the rotated vector, ω_i as the remaining angle and d_i as the decision variable. i is the iteration number with $i \in \{0, 1, ..., n-1\}$.

Appendix E
Formal definitions

This appendix lists all formal definitions of mentioned algorithms.

Algorithm 1: Coherent De-Dispersion with cdmt, slightly rewritten from the original for more clearance

cdmt (DM_{set}, S_{in});
Input : Set of possible DM values, input signal of complex values of length N
Output: $len(DM_{set})$ signals with de-dispersion removed with certain DM
$N_N \leftarrow N_{bin} - n_d$
$B \leftarrow n_c$ pieces of length N_{bin} from S_{in} overlapped with n_d
foreach $B_i \in \{B\}$ **do**
  $B_i \leftarrow FFT(B_i)$
end
$S_{out} \leftarrow \emptyset$
foreach $DM \in DM_{set}$ **do**
  $S_{out}^{DM} \leftarrow \emptyset$
  foreach $B_i \in \{B\}$ **do**
    $chrp_i \leftarrow CHIRP(DM)$
    $B_i^{DM} \leftarrow B_i \times chrp_i$
    $B_i^{DM} \leftarrow IFFT(B_i^{DM})$
    $B_i^{DM} \leftarrow B_i^{DM}$ minus n_d samples
    $S_{out}^{DM} \leftarrow \{S_{out}^{DM}\} \bigcup \{B_i^{DM}\}$
  end
  $S_{out} \leftarrow \{S_{out}\}$ append $\{S_{out}^{DM}\}$
end

Algorithm 2: Algorithm to generate matrix of input indices

```
input_indices (N,P);
Input : FFT length N and parallel ports P
Output: matrix of input indices M_out
M ← ∅
for i ∈ {0,1,...,log2(P)−1} do
    B1 ← binary(i)
    B2 ← reversed(B1) ≪ S − log2(P)
    M append {int(B1), int(B1)+1, ..., int(B2)}
end
```

Algorithm 3: algorithm to generate input pattern for stage s and P parallel inputs, please note it consists of main routine *connect_pattern* and sub routine *connect_pattern_base*

```
connect_pattern (s,P);
Input : stage s and parallel ports P
Output: Vector of input indexes ptrn
if s < log2(P)−1 then
    ptrn ← ∅
    for i ∈ {0, 2^s, ..., P−1} do
        ptrn ← ptrn ⋃ connect_pattern_base(i, 2^s)
    end
else
    ptrn ← connect_pattern_base(0,P)
end
connect_pattern_base (b,T);
Input : base b and width T
Output: Vector of input indexes ptrn
ptrn ← ∅
for p ∈ {0,1,...,T} do
    if p < T/2 then
        ptrn ← ptrn ⋃ {b + 2·p}
    else
        ptrn ← ptrn ⋃ {b + (p − ⌊T/2⌋)·2}
    end
end
```

$$S_0 \vee S_1 \begin{cases} I_0 \to L_1 \\ I_1 \to L_2 \\ L_1 \to O_0 \\ L_2 \to O_1 \end{cases}, S_2 \begin{cases} I_0 \to O_1 \\ I_1 \to L_2 \\ L_1 \to O_0 \\ L_2 \to L_1 \end{cases} \tag{15}$$

APPENDIX F
RADIX-2^2

The series of stages of radix-2 can be further optimized by rewriting the twiddle factors by splitting all non trivial rotations ϕ into a trivial and non trivial part ϕ' and rewrite resulting expression as follows:

$$Ae^{-j\frac{2\pi}{N}\phi'} \pm Be^{-j\frac{2\pi}{N}(\phi'+N/4)} = [A \pm (-j)B] \cdot e^{-j\frac{2\pi}{N}\phi'} \tag{16}$$

Algorithm 4: algorithm to generate rotation matrix M_P^R

```
twiddle_rotations (s,N,P);
Input : stagenumber s, FFT length N > 0, parallel
        ports P ∈ {4,8,16}
Output: Rotatation angles matrix M_P^R
M_P^R ← ∅
if s = 1 then
    l ← {0,1,...,⌊P/4⌋}
    l ← even(l) ⋃ odd(l)
    for α ∈ {0,1,...,N/P} do
        B ← ∅
        for b ∈ {0,1,...,P/4 − 1} do
            B append (α + b * ⌊N/P⌋) · {0,2,1,3}
        end
        Reshuffle B with indices according to l
        M_P^R append B
    end
else
    τ ← 0
    for i ∈ {0,1,...,N/P} do
        T ← τ × 4^⌊(s−1)/2⌋ · {0,2,1,3}
        C ← P/4 times copy of T concatenated
        M_P^R append C
        if τ ≥ N/4 then
            τ ← 1
        else
            τ ← τ + 1
        end
    end
end
```

in which A and B are the inputs of the butterfly and $\phi' = \phi \ mod \ N/4$. In the resulting expression the bracketed part has only a trivial rotation and then this part is rotated with a non trivial angle. In practice this corresponds to moving the non trivial part to the next stage for all odd stages, as depicted in figure 2 [10]. This method reduces the total number of non trivial rotations in the flow graph on the expense of a slighter more complex control circuit.

APPENDIX G
XILINX FPGA STRUCTURE

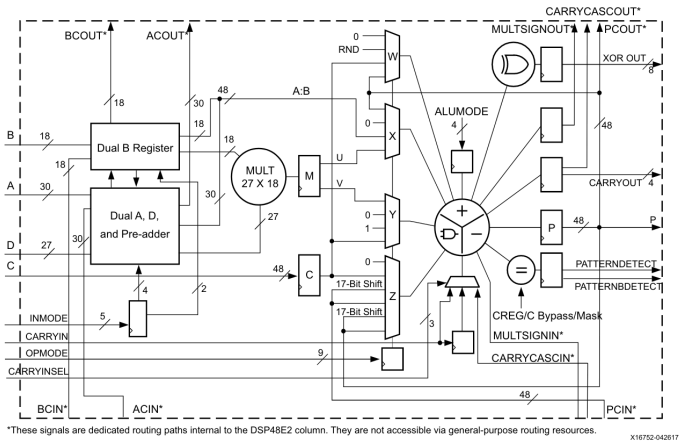


Fig. 19: typical xilinx DSP slice schematic, taken from [26]

APPENDIX H
OPERATIONAL INTENSITY OF FFT

To calculate the Arithmetic Intensity of a FFT first the number operations for FFT size N is approximated. Taking the de facto benchmarking standard reference of FFTW [28] this corresponds to:

$$num\_operations = 5 \cdot N \cdot \log_2(N), \tag{17}$$

with N the size of the FFT. This is the same for both GPU and FPGA.

For GPU the CUDA library is used for programming. Due to hardware constraints CUDA will copy data from host to device and back approximately once every three stages [18]. Furthermore all data types are at least 32-bit floating point, this results in using 4 bytes for both real and imaginary part. Lastly there is a in and output for FFT of approximately N times 2 bytes each time, when taking the input data characteristic of [3].

Taking this together results in the expression

$$mem\_usage = \lceil \frac{S}{3} \rceil \cdot 8 \cdot N + 4 \cdot N \tag{18}$$

here S is number of stages and N is the size of the FFT. For FPGA everything can be streamed through the architecture thus resulting in only the in and output memory transfer of a total of $4 \cdot N$.

When filling in the parameters of $S = 16, N = 2^{16}$ of [3] and calculate the arithmetic intensity ai it is found that $ai_{gpu} = 1.54$ and $ai_{FPGA} = 20$

APPENDIX I
BASSA BENCHMARK RESULT

For reading convenience figure 6 of [3] is shown here, which shows the results of their benchmark.

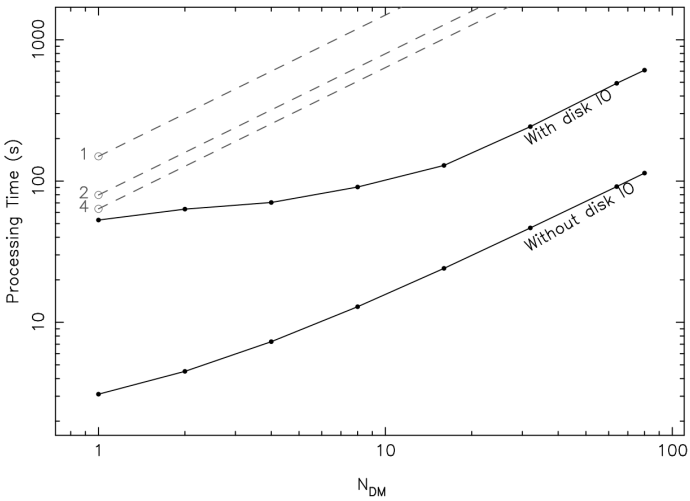


Fig. 20: figure 6 from [3]