EINDHOVEN UNIVERSITY OF TECHNOLOGY

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

Graduation project

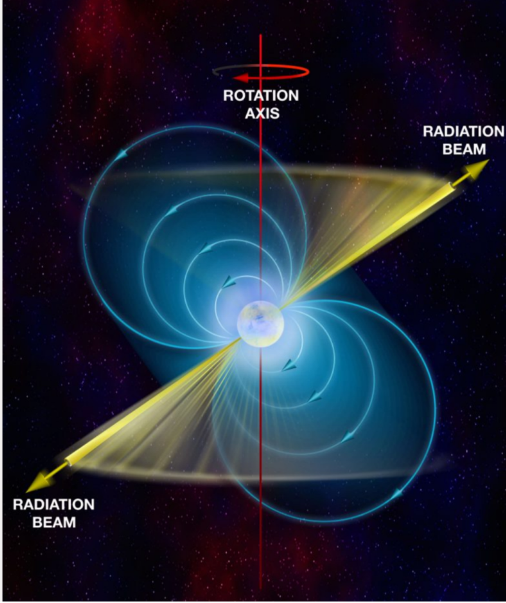
Frank Boerman

Electrical Engineering

Radio Pulsars

"rapidly rotating heavily magnetised neutron stars"
slowly losing rotation speed $\rightarrow$ energy loss $\dot{E}$

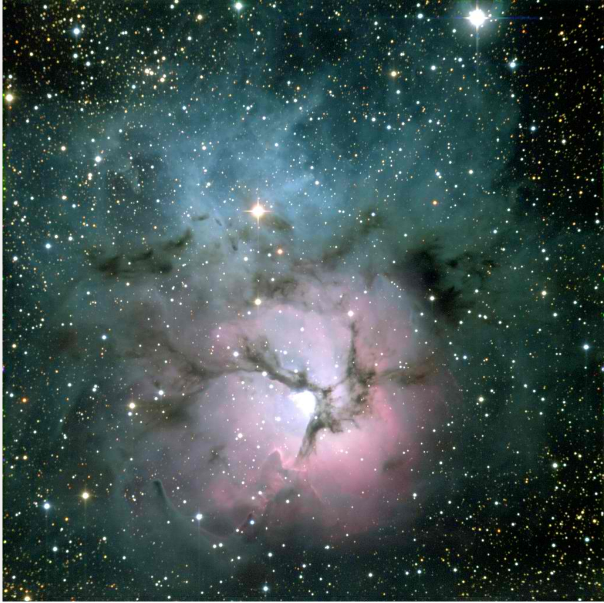
small fraction $\rightarrow$ radio emission reaching Earth
Rotating radiation beam $\rightarrow$ "blinking" pulse effect



The InterStellar Medium (ISM)

consist of cold ionised plasma
Four distinct effects:

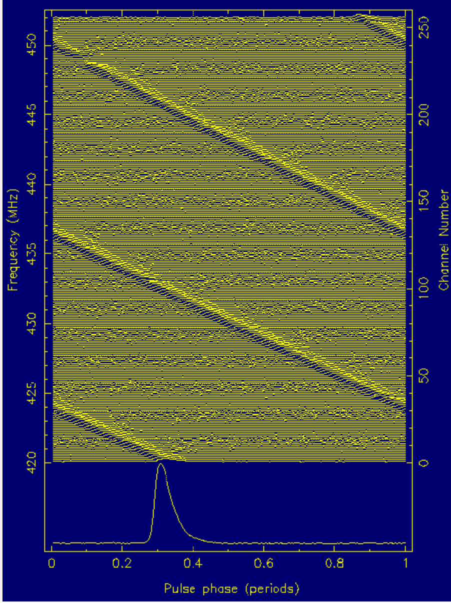
- Faraday Rotation
- Scintillation
- Scattering
- **Frequency Dispersion**



TU/e

Frequency Dispersion

"the effect in which signals with different wavelength have different propagation speeds through a non-vacuum medium"

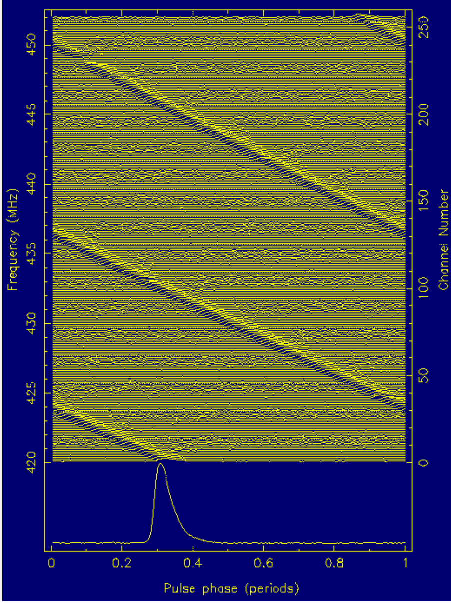


TU/e

Frequency Dispersion

"the effect in which signals with different wavelength have different propagation speeds through a non-vacuum medium"

different frequencies within the signal will slowly shift in time relative to each other



Frequency De-Dispersion

Split in small subbands and apply one of two options:

Frequency De-Dispersion

Split in small subbands and apply one of two options:
Incoherent, simple timeshift:

$$\Delta t = 4.15 \times 10^{-6} \times DM \times (f_1^{-2} - f_2^{-2}) \tag{1}$$

Frequency De-Dispersion

Split in small subbands and apply one of two options:
Incoherent, simple timeshift:

$$\Delta t = 4.15 \times 10^{-6} \times DM \times (f_1^{-2} - f_2^{-2}) \tag{1}$$

Coherent, phase only filter with transfer function:

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

Frequency De-Dispersion

Split in small subbands and apply one of two options:
Incoherent, simple timeshift:

$$\Delta t = 4.15 \times 10^{-6} \times DM \times (f_1^{-2} - f_2^{-2}) \tag{1}$$

Coherent, phase only filter with transfer function:

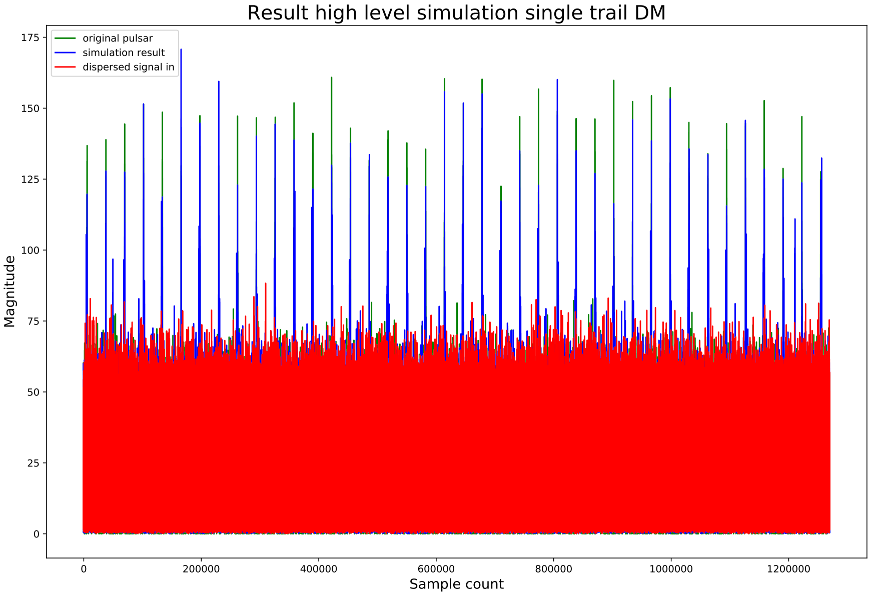
$$H(f + f_0) = exp\left[\frac{2\pi \cdot i \cdot f^2 \cdot k_{DM} \cdot DM}{f_0^2(f + f_0)}\right] \tag{2}$$

with **Dispersion Measure**:

$$DM = \int_0^d \eta_e(l)dl \tag{3}$$

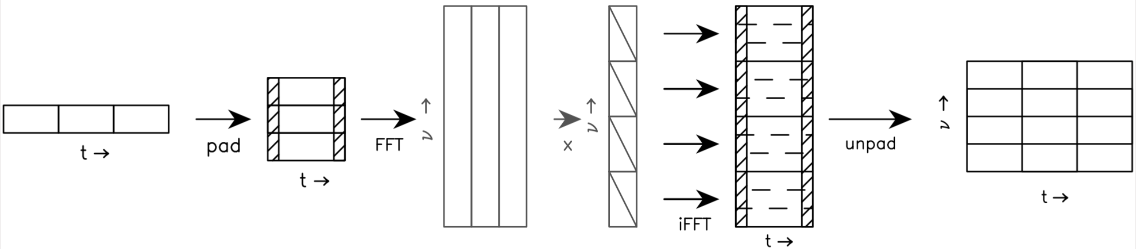
(De)Dispersion visualized

Green:
pulsar signal
Red:
telescope data
-> no longer peaks!
Blue:
example output
-> recovered peaks



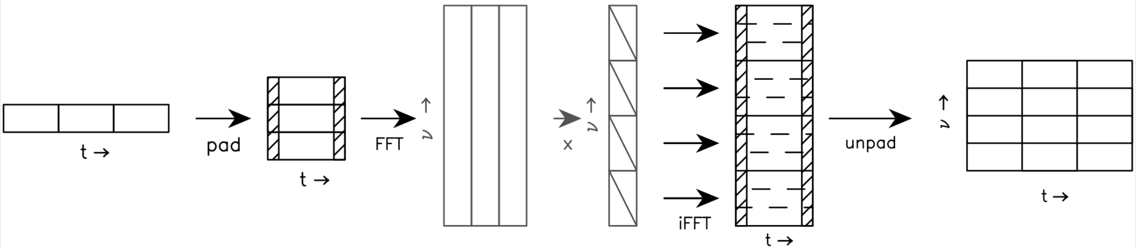
CDMT (Bassa et al, 2016)

coherent dispersion measure trials



CDMT (Bassa et al, 2016)

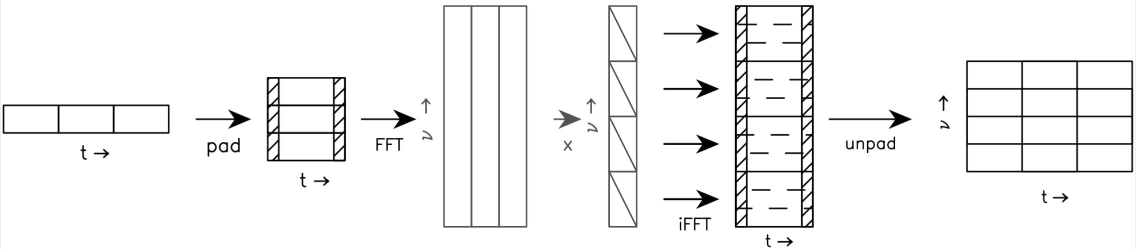
coherent dispersion measure trials



Filter multiplication in frequency domain →

CDMT (Bassa et al, 2016)

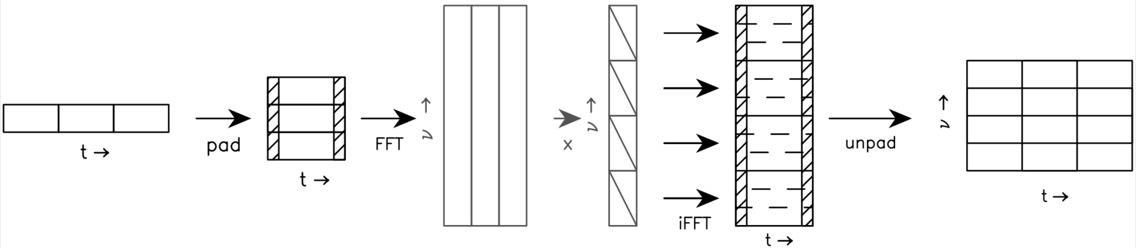
coherent dispersion measure trials



Filter multiplication in frequency domain → **Fourier Transform (FFT)**

CDMT (Bassa et al, 2016)

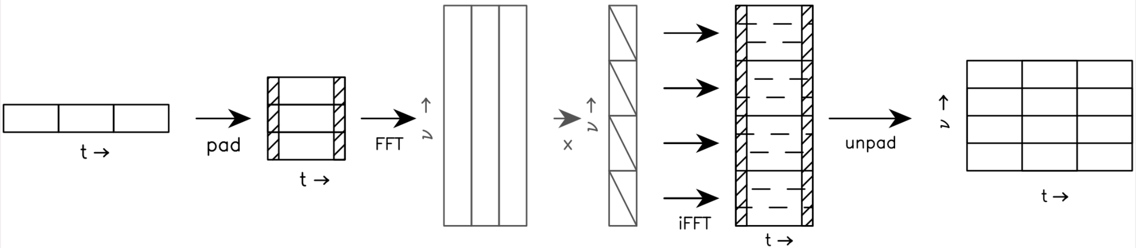
coherent dispersion measure trials



Filter multiplication in frequency domain → **Fourier Transform (FFT)**
State of the art implemented on **GPU** with many **DM trails**

CDMT (Bassa et al, 2016)

coherent dispersion measure trials



Filter multiplication in frequency domain → **Fourier Transform (FFT)**
State of the art implemented on **GPU** with many **DM trails**
Not real time -> large intermediate storage needed

Problem Statement

To recap:

Problem Statement

To recap:
Pulsar signals are smeared beyond recognition

Problem Statement

To recap:
Pulsar signals are smeared beyond recognition

Intensive parallel recovery operation is needed **In Frequency Domain**

Problem Statement

To recap:
Pulsar signals are smeared beyond recognition

Intensive parallel recovery operation is needed **In Frequency Domain**

State of the art is slow -> Not real time -> Large intermediate storage needed

Problem Statement

To recap:
Pulsar signals are smeared beyond recognition

Intensive parallel recovery operation is needed **In Frequency Domain**

State of the art is slow -> Not real time -> Large intermediate storage needed

Can we process faster to real time?

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

$$I_O = \frac{number\_of\_operations}{amount\_of\_DRAM\_traffic(input + output)[bytes]} \tag{4}$$

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

$$I_O = \frac{number\_of\_operations}{amount\_of\_DRAM\_traffic(input + output)[bytes]} \tag{4}$$

For FFT with $N = 2^{16}$ (so 16 stages)

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

$$I_O = \frac{number\_of\_operations}{amount\_of\_DRAM\_traffic(input + output)[bytes]} \tag{4}$$

For FFT with $N = 2^{16}$ (so 16 stages)
GPU: $I_O \approx 1.5$
FPGA: $I_O \approx 20$
Why GPU inefficient? (from Govindaraju et al, 2008)

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

$$I_O = \frac{number\_of\_operations}{amount\_of\_DRAM\_traffic(input + output)[bytes]} \tag{4}$$

For FFT with $N = 2^{16}$ (so 16 stages)
GPU: $I_O \approx 1.5$
FPGA: $I_O \approx 20$
Why GPU inefficient? (from Govindaraju et al, 2008)

- all data minimum 32-bit (4 bytes) floating point

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

$$I_O = \frac{number\_of\_operations}{amount\_of\_DRAM\_traffic(input + output)[bytes]} \tag{4}$$

For FFT with $N = 2^{16}$ (so 16 stages)
GPU: $I_O \approx 1.5$
FPGA: $I_O \approx 20$
Why GPU inefficient? (from Govindaraju et al, 2008)

- all data minimum 32-bit (4 bytes) floating point
- approximately every three stages full read and write

Some terminology

Operational intensity I_O = amount of compute / unit DRAM traffic

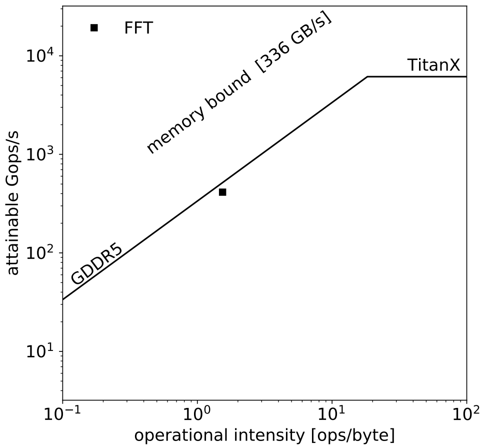
$$I_O = \frac{number\_of\_operations}{amount\_of\_DRAM\_traffic(input + output)[bytes]} \quad (4)$$

For FFT with $N = 2^{16}$ (so 16 stages)
GPU: $I_O \approx 1.5$
FPGA: $I_O \approx 20$
Why GPU inefficient? (from Govindaraju et al, 2008)

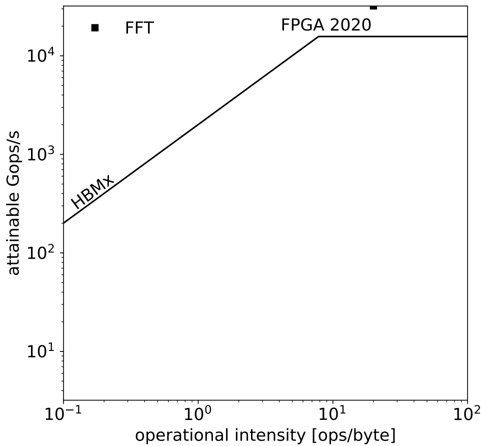
- all data minimum 32-bit (4 bytes) floating point
- approximately every three stages full read and write
- **intermediate results are saved many times!**

RoofLines

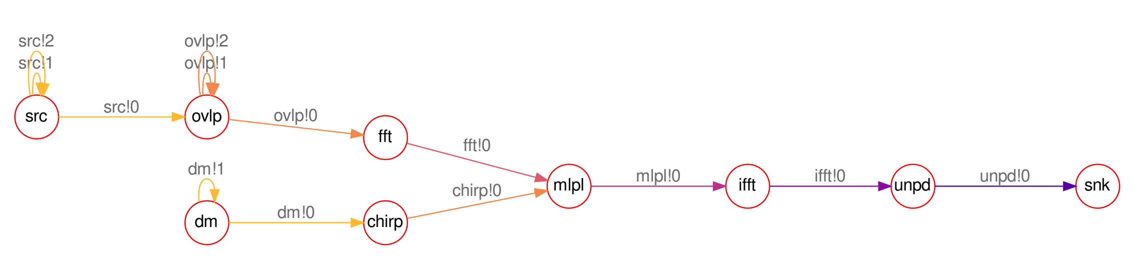
GPU



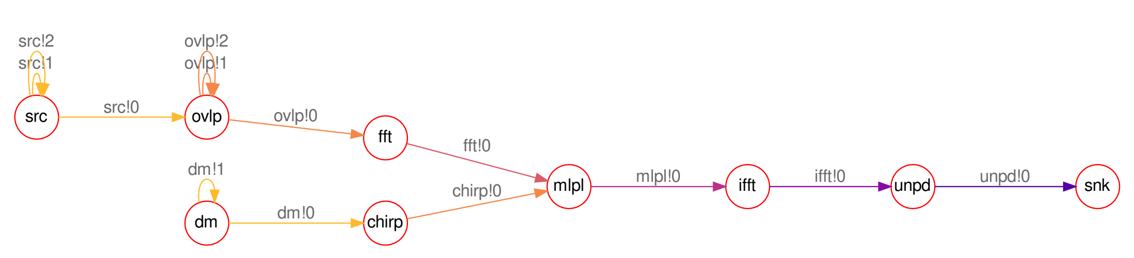
FPGA



De-Dispersion flow (*cdmt*)

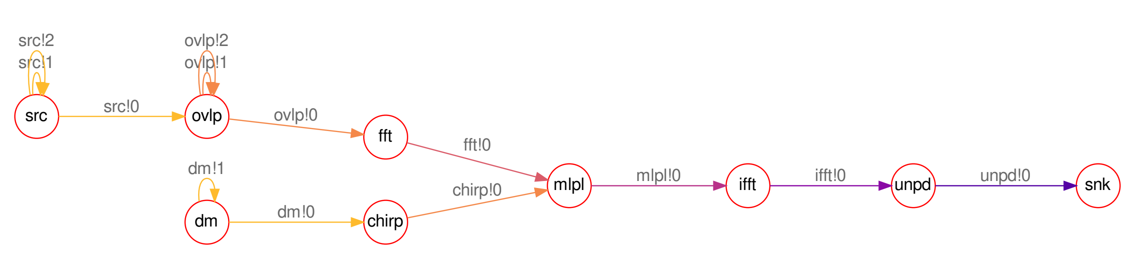


De-Dispersion flow (*cdmt*)



Dominant complexity -> **Crossing time-frequency boundary**

De-Dispersion flow (*cdmt*)

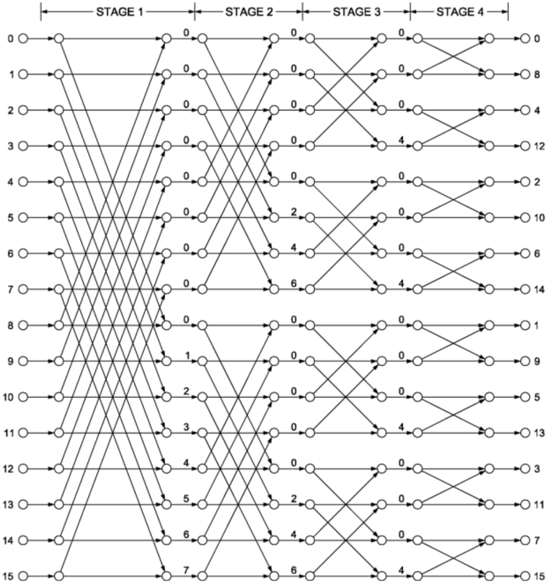


Dominant complexity -> **Crossing time-frequency boundary**

For FPGA: the whole trail fully on chip!

The DFT

Important:
N number of samples in single FFT
across $S = \log_2(N)$ stages

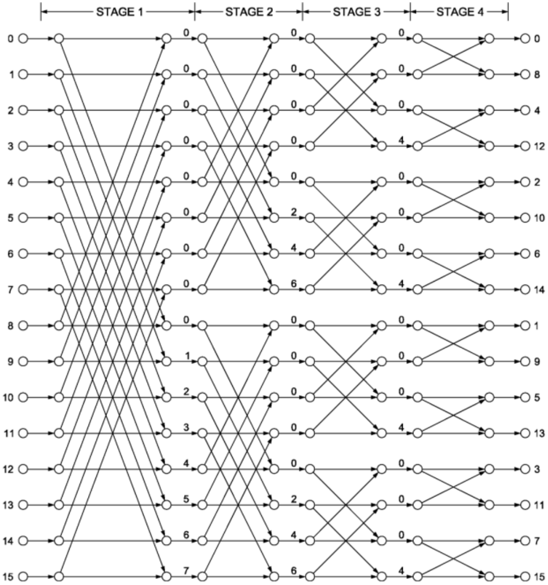


The DFT

Important:
N number of samples in single FFT
across $S = \log_2(N)$ stages

"butterfly" addition and subtraction
"twiddle" rotating factor

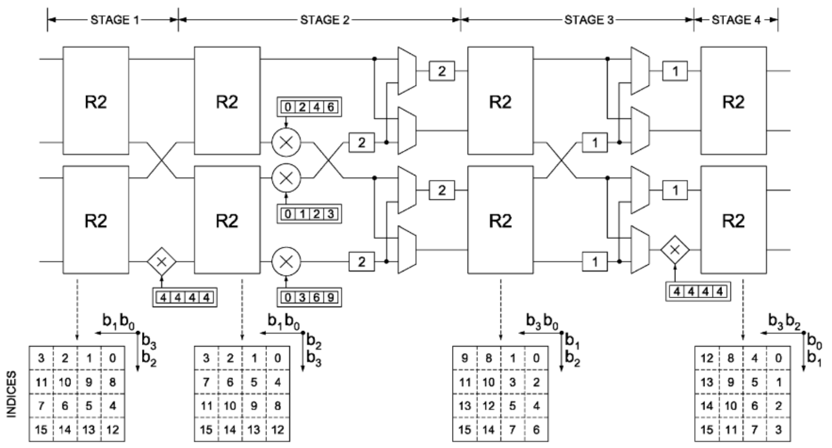
optimized algorithm is called
Fast **F**ourier **T**ransform (**FFT**)



FFT: Desired Properties

- High FFT size N
- High throughput, high parallel samples P
- Small buffers (memory hardware is expensive!)
- easily scalable

DFT Architecture (Garrido et al, 2013)



LOFAR use case: how many trails can we fit?

cdmt used by **LO**w **F**requency **AR**ray

LOFAR use case: how many trails can we fit?

cdmt used by **LO**w **F**requency **AR**ray

- 200 dual-polarized sub-bands with a bandwidth of 195.32 kHz each
- processed by 23 nodes of 4 NVIDIA TITAN GPU's per node
- grand total of 80 DM trails for whole system

LOFAR use case: how many trails can we fit?

cdmt used by **LO**w **F**requency **AR**ray

- 200 dual-polarized sub-bands with a bandwidth of 195.32 kHz each
- processed by 23 nodes of 4 NVIDIA TITAN GPU's per node
- grand total of 80 DM trails for whole system
- $2 \cdot 200 \cdot 195.32 \times 10^3 = 78.128 \times 10^6$ samples per second

LOFAR use case: how many trails can we fit?

- $2 \cdot 200 \cdot 195.32 \times 10^3 = 78.128 \times 10^6$ samples per second

LOFAR use case: how many trails can we fit?

- $2 \cdot 200 \cdot 195.32 \times 10^3 = 78.128 \times 10^6$ samples per second
- Can we fit this realtime? yes!

LOFAR use case: how many trails can we fit?

- $2 \cdot 200 \cdot 195.32 \times 10^3 = 78.128 \times 10^6$ samples per second
- Can we fit this realtime? yes!
- Example synthesis: $f = 143Mhz$ for $P = 16$ (Garrido et al, 2014)
- Throughput of $143 \times 10^6 \cdot 16 = 2.29 \times 10^9$

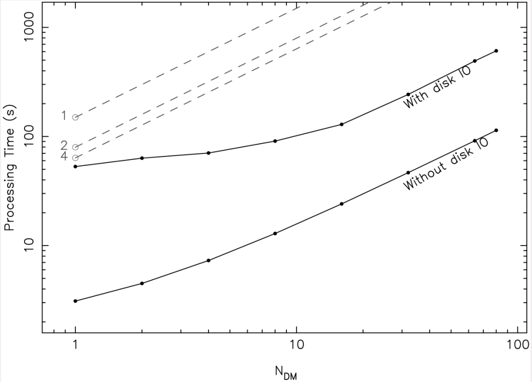
LOFAR use case: how many trails can we fit?

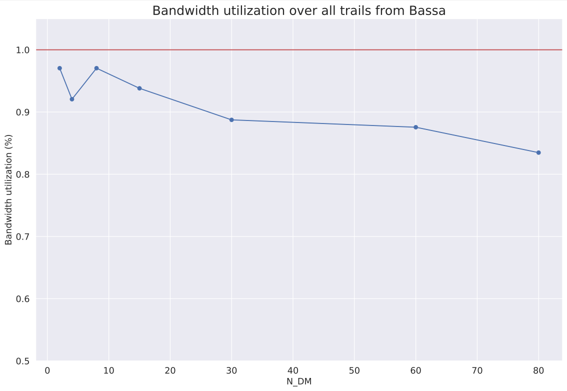
- $2 \cdot 200 \cdot 195.32 \times 10^3 = 78.128 \times 10^6$ samples per second
- Can we fit this realtime? yes!
- Example synthesis: $f = 143Mhz$ for $P = 16$ (Garrido et al, 2014)
- Throughput of $143 \times 10^6 \cdot 16 = 2.29 \times 10^9$
- $\dfrac{2.29 \times 10^9}{78.128 \times 10^6} = 29$ trails fit realtime!

TU/e

cdmt benchmark

lets look in detail of the whole dispersion





Numerical Comparison

GPU per polarization pair:

Numerical Comparison

GPU per polarization pair:

- Input: 2 complex samples of each 2 bytes: 4 bytes
- Output: 1 byte *Stokes parameter*

Numerical Comparison

GPU per polarization pair:

- Input: 2 complex samples of each 2 bytes: 4 bytes
- Output: 1 byte *Stokes parameter*
- FFT intermediate every 3 stages (this case: $\lceil 16/3 \rceil = 6$ times):

Numerical Comparison

GPU per polarization pair:

- Input: 2 complex samples of each 2 bytes: 4 bytes
- Output: 1 byte *Stokes parameter*
- FFT intermediate every 3 stages (this case: $\lceil 16/3 \rceil = 6$ times):
  - Write 2 polarizations of complex samples of each $2 * 4$ bytes: 16 bytes
  - Read 2 polarizations of complex samples of each $2 * 4$ bytes: 16 bytes

Numerical Comparison

GPU per polarization pair:

- Input: 2 complex samples of each 2 bytes: 4 bytes
- Output: 1 byte *Stokes parameter*
- FFT intermediate every 3 stages (this case: $\lceil 16/3 \rceil = 6$ times):
  - Write 2 polarizations of complex samples of each $2 * 4$ bytes: 16 bytes
  - Read 2 polarizations of complex samples of each $2 * 4$ bytes: 16 bytes
- Grand total: $4 + 1 + 6 * (16 + 16) = 197$ bytes

Numerical Comparison (cont.)

Numerical Comparison (cont.)

FPGA per polarization pair:

- Input: 2 complex samples of each 2 bytes: 4 bytes
- Output: 1 byte *Stokes parameter*

Numerical Comparison (cont.)

FPGA per polarization pair:

- Input: 2 complex samples of each 2 bytes: 4 bytes
- Output: 1 byte *Stokes parameter*
- Grand total: $4 + 1 = 5$ bytes
- $\frac{197}{5} \approx 40$ times improvement possible!

What is DataFlow programming?

model complex systems as independent actors with communication in between

What is DataFlow programming?

model complex systems as independent actors with communication in between

now also use for actual programming!

What is DataFlow programming?

model complex systems as independent actors with communication in between

now also use for actual programming!

Implemented in library called **StaccatoLab**

What is DataFlow programming?

model complex systems as independent actors with communication in between

now also use for actual programming!

Implemented in library called **StaccatoLab**

Consists of:

- Nodes (with output function *fo*)
- Edges
- Tokens

What is DataFlow programming?

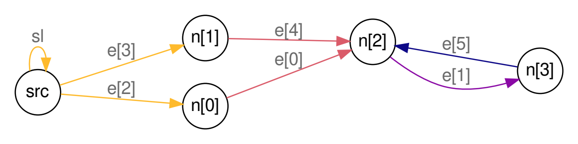
model complex systems as independent actors with communication in between

now also use for actual programming!

Implemented in library called **StaccatoLab**

Consists of:

- Nodes (with output function *fo*)
- Edges
- Tokens

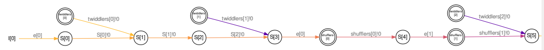


Implementation FFT

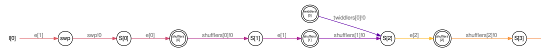
Conversion to dataflow step by step

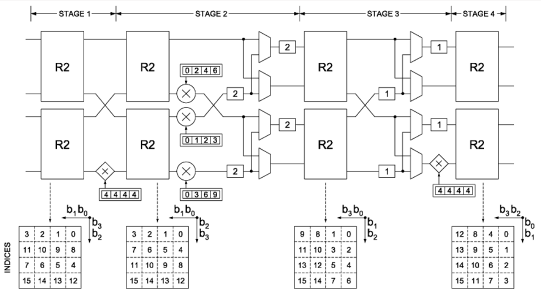
Dataflow implementation

FFT:



IFFT:





The tokens: SIMD

Put computational complexity in node output function *fo*

The tokens: SIMD

Put computational complexity in node output function *fo*

Let StaccatoLab do the optimization!

Prevent forcing long wiring

The tokens: SIMD

Put computational complexity in node output function *fo*

Let StaccatoLab do the optimization!

Prevent forcing long wiring

Every token is a vector length P

The tokens: SIMD

Put computational complexity in node output function *fo*

Let StaccatoLab do the optimization!

Prevent forcing long wiring

Every token is a vector length P

$$\vec{V^i} : \begin{bmatrix} s^i_0 \\ s^i_1 \\ s^i_{\dots} \\ s^i_{P-1} \end{bmatrix} \rightarrow \vec{V^o} : \begin{bmatrix} s^o_0 \\ s^o_1 \\ s^o_{\dots} \\ s^o_{P-1} \end{bmatrix}$$

The tokens: SIMD

Put computational complexity in node output function *fo*

Let StaccatoLab do the optimization!

Prevent forcing long wiring

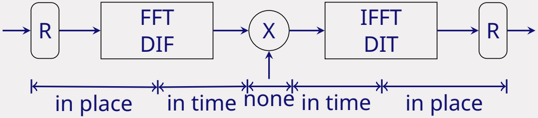
Every token is a vector length P

$$\vec{V^i} : \begin{bmatrix} s^i_0 \\ s^i_1 \\ s^i_{\dots} \\ s^i_{P-1} \end{bmatrix} \rightarrow \vec{V^o} : \begin{bmatrix} s^o_0 \\ s^o_1 \\ s^o_{\dots} \\ s^o_{P-1} \end{bmatrix}$$

Requires every action to be compatible!

Input/output ordering

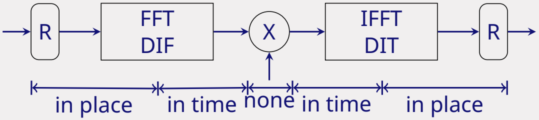
Property of FFT: *bit reversed order*
indices of output samples are bit reversed
example: 0010 (2) -> 0100 (4)



Input/output ordering

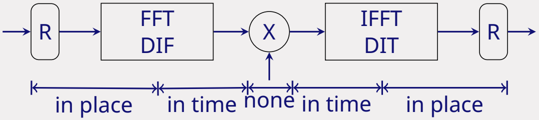
Property of FFT: *bit reversed order* indices of output samples are bit reversed
example: 0010 (2) -> 0100 (4)

Division **i**n **F**requency vs
Division **i**n **T**ime
→ inversed flow diagram



Input/output ordering

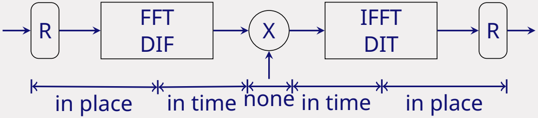
Property of FFT: *bit reversed order*
indices of output samples are bit reversed
example: 0010 (2) -> 0100 (4)



Division **i**n **F**requency vs
Division **i**n **T**ime
→ inversed flow diagram
→ DIF natural input, bit reversed output

Input/output ordering

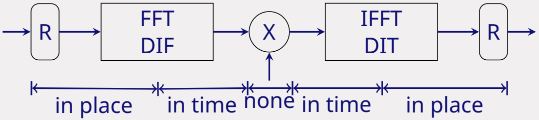
Property of FFT: *bit reversed order*
indices of output samples are bit reversed
example: 0010 (2) -> 0100 (4)



Division **i**n **F**requency vs
Division **i**n **T**ime
$\rightarrow$ inversed flow diagram
$\rightarrow$ DIF natural input, bit reversed output
$\rightarrow$ DIF bit reversed input, natural output

Input/output ordering

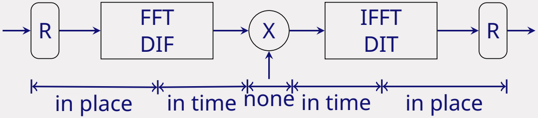
Property of FFT: *bit reversed order*
indices of output samples are bit reversed
example: 0010 (2) -> 0100 (4)



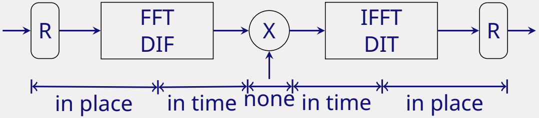
Division **i**n **F**requency vs
Division **i**n **T**ime
$\rightarrow$ inversed flow diagram
$\rightarrow$ DIF natural input, bit reversed output
$\rightarrow$ DIF bit reversed input, natural output

Using both saves a reorder!

Input/output ordering (cont.)

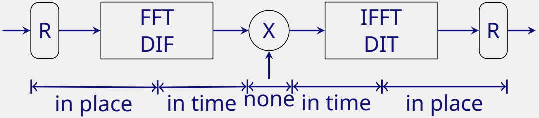


Input/output ordering (cont.)



The architecture does not bit reverse all samples!

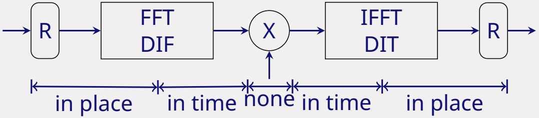
Input/output ordering (cont.)



The architecture does not bit reverse all samples!

Only ones with indices different in the lower $\log_2(N) - \log_2(P)$ bits

Input/output ordering (cont.)

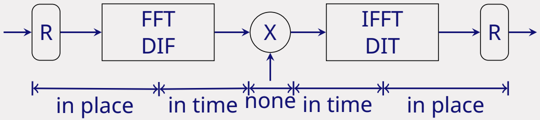


The architecture does not bit reverse all samples!

Only ones with indices different in the lower $\log_2(N) - \log_2(P)$ bits

Separate block needed for first $\log_2(P)$ bits

Input/output ordering (cont.)



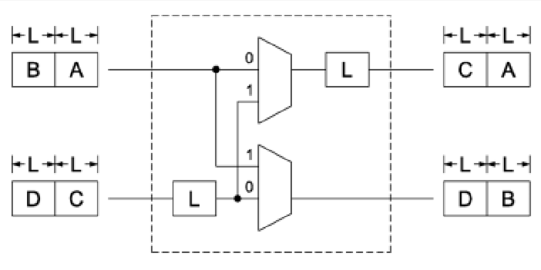
The architecture does not bit reverse all samples!

Only ones with indices different in the lower $\log_2(N) - \log_2(P)$ bits

Separate block needed for first $\log_2(P)$ bits

The first only changes samples over input ports (vertical), the second also reorders in time (horizontal)

Shuffler blocks

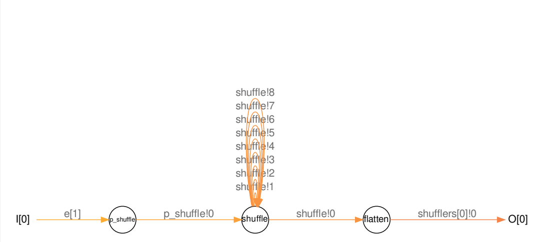


Continues the reordering of previous step
reorders samples across time and place, bufferlength L

there are always $\log_2(N) - \log_2(P)$ shuffles

Shuffler blocks (cont.)

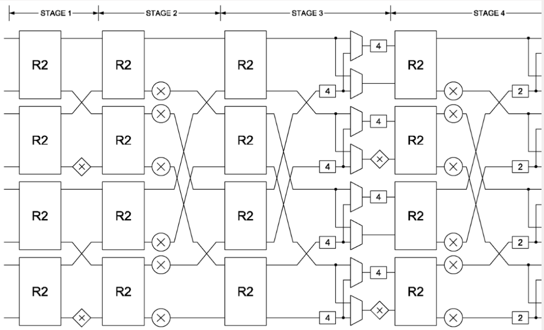
example of $P = 8$



Buffers per edge(P) -> take apart the SIMD!

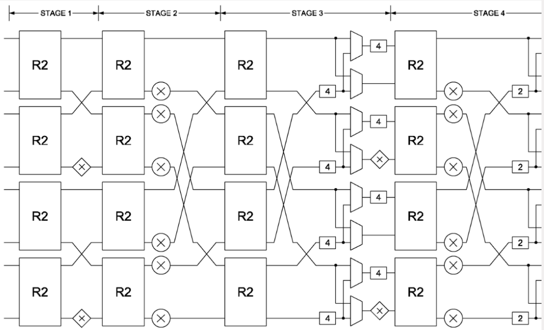
Multiplexor -> Finite State Machine (thus *Cyclo Static Dataflow*)

Connection Pattern



Depending on P: implemented as single vector in token

Connection Pattern



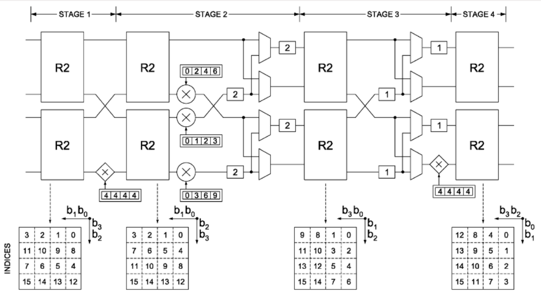
Depending on P: implemented as single vector in token

In dataflow:
Instead of wiring -> fixed reorder of samples within the vector

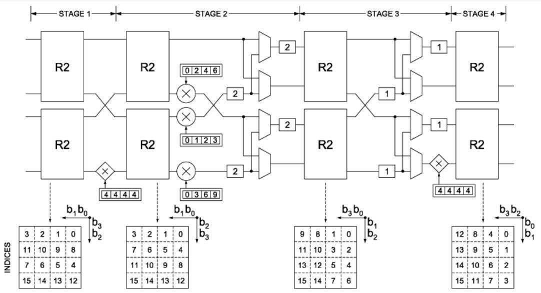
hardcoded precomputed ordering per stage

Butterfly (R2)

R2 block defined on pairs of 2 samples



Butterfly (R2)



R2 block defined on pairs of 2 samples

in dataflow split input vector in pairs and apply:

$$
\begin{aligned}
O_0 &= I_0 + I_1 \\
O_1 &= I_0 - I_1
\end{aligned} \tag{5}
$$

Twiddle rotation in Architecture

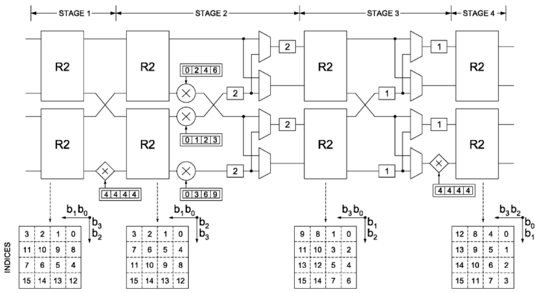
Twiddle factors not discussed at all in original architecture!

Rotations expressed as integer ϕ

$$W_N^{nk} = W_N^{\phi} = e^{\dfrac{-j \cdot 2\pi \cdot \phi}{N}} \tag{6}$$

Trivial rotations: $\{0, N/4, N/2, 3N/4\}$

Trivial: $\rightarrow \{0, 4, 8, 12\}$



Realizing non-trivial rotations

example of $N = 64$ $P = 16$ for integer

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

Realizing non-trivial rotations

example of $N = 64$ $P = 16$ for integer

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

Based on a pattern:

$$(0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\})$$
$$+ \alpha_{row}$$

Realizing non-trivial rotations

example of $N = 64$ $P = 16$ for integer

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

Based on a pattern:

$$(0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\})$$
$$+ \alpha_{row}$$

Nodes:

- $base_\alpha$

Realized as:

- CORDIC: hardware

Realizing non-trivial rotations

example of $N = 64$ $P = 16$ for integer

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

Based on a pattern:

$$(0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\})$$
$$+ \alpha_{row}$$

Nodes:

- $base_\alpha$
- $R_{n\alpha}$ (with $n \in \{2, 3\}$)

Realized as:

- CORDIC: hardware
- Rotation matrix using goniometric properties

Realizing non-trivial rotations

example of $N = 64$ $P = 16$ for integer

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

Based on a pattern:

$$(0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\})$$
$$+ \alpha_{row}$$

Nodes:

- $base_\alpha$
- $R_{n\alpha}$ (with $n \in \{2, 3\}$)
- $R_{N//P}$

Realized as:

- CORDIC: hardware
- Rotation matrix using goniometric properties
- Rotation matrix precomputed

Rotator Nodes

CORDIC: hardware to approximate any angle, used for base step

Rotator Nodes

CORDIC: hardware to approximate any angle, used for base step

$R_{N//P}$ precomputed rotation matrix (exact rotation!)

$$R(\alpha) \cdot V_C = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{7}$$

Rotator Nodes

CORDIC: hardware to approximate any angle, used for base step
$R_{N//P}$ precomputed rotation matrix (exact rotation!)

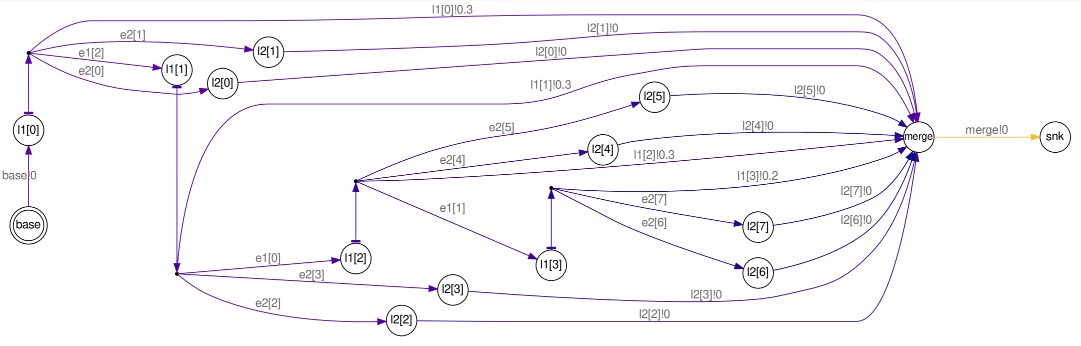
$$R(\alpha) \cdot V_C = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{7}$$

$R_{n\alpha}$ make use of goniometric identity in rotation matrix:

$$\begin{aligned} \begin{bmatrix} x'_{2\alpha} \\ y'_{2\alpha} \end{bmatrix} &= \begin{bmatrix} \cos(2\alpha_0) & -\sin(2\alpha_0) \\ \sin(2\alpha_0) & \cos(2\alpha_0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x^2 - y^2 \\ 2 \cdot x \cdot y \end{bmatrix} \\ \begin{bmatrix} x'_{3\alpha} \\ y'_{3\alpha} \end{bmatrix} &= \begin{bmatrix} \cos(3\alpha_0) & -\sin(3\alpha_0) \\ \sin(3\alpha_0) & \cos(3\alpha_0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \cdot x^3 - 3 \cdot x \\ 3 \cdot y - 4 \cdot y^3 \end{bmatrix} \end{aligned} \tag{8}$$

Twiddle generator Dataflow implementation $s = 1$

Everything together produces this work of art:



the Inverse Fourier Transform

the DFT definition:

$$X[k] = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{9}$$

inverse DFT (IFFT):

$$x_n = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn} \tag{10}$$

the Inverse Fourier Transform

the DFT definition:

$$X[k] = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{9}$$

inverse DFT (IFFT):

$$x_n = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn} \tag{10}$$

rewriting in eachother:

$$\mathcal{F}^{-1}(\vec{x}) = \frac{1}{N} \cdot swap(\mathcal{F}(swap(\vec{x}))) \tag{11}$$

with $swap : a + bi \to b + ai$
or in math:

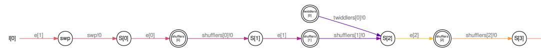
$$swap(x_n) = ix_n^* \tag{12}$$

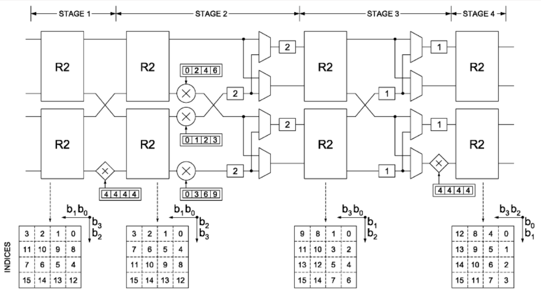
can now express the inverse as the original with simple swap

Dataflow Graph final result (again)

FFT:

IFFT:

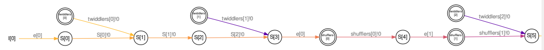


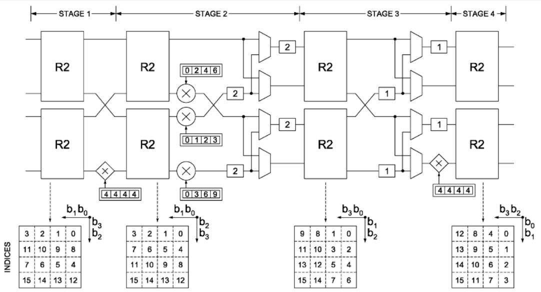


Library created to generate design, fully scalable through *N* and *P*

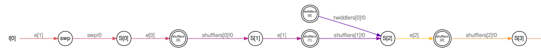
Dataflow Graph final result (again)

FFT:





IFFT:



Library created to generate design, fully scalable through *N* and *P*
All arithmetic internally done in fixed point with *Z* bits per sample part

Testing Library

Test the FFT with its main properties, from industry standard FFTW library:

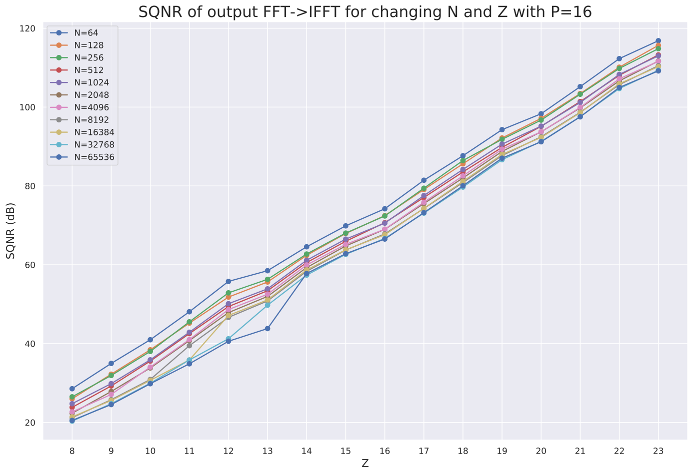
Testing Library

Test the FFT with its main properties, from industry standard FFTW library:

- Unit pulse input $\rightarrow$ "flat" output: tests butterfly without twiddles
- Linearity: $\mathcal{F}(\vec{V_A} + \vec{V_B}) = \mathcal{F}(\vec{V_A}) + \mathcal{F}(\vec{V_B})$
- Time shift of D samples: $\mathcal{F}(\vec{V_I^D}) = \left[e^{\left(\dfrac{-2j\pi D}{N} n\right)} \right] \cdot \mathcal{F}\left(\vec{V_I}\right)$
- Alternative $+2^Z$, -2^Z to test for overflow
- White noise: general data test

All simulation only

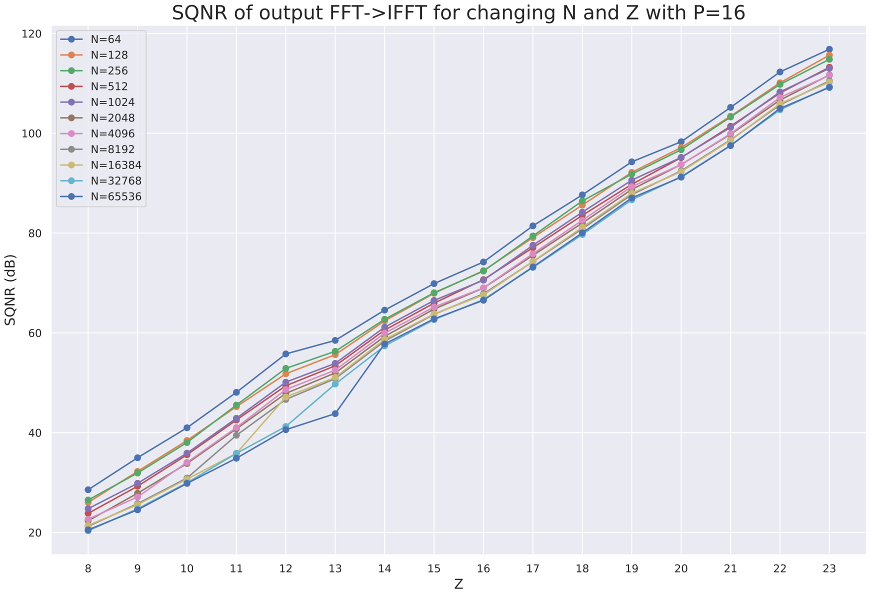
FFT -> IFFT



SQNR of output FFT->IFFT for changing N and Z with P=16
N=64
N=128
N=256
N=512
N=1024
N=2048
N=4096
N=8192
N=16384
N=32768
N=65536
SQNR (dB)
120
100
80
60
40
20
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
Z

FFT -> IFFT

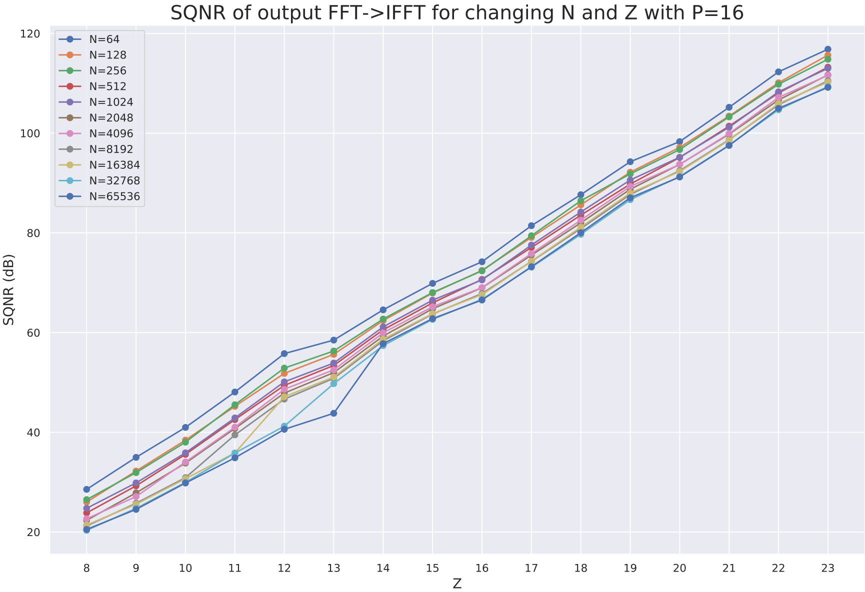
What do we see?



FFT -> IFFT

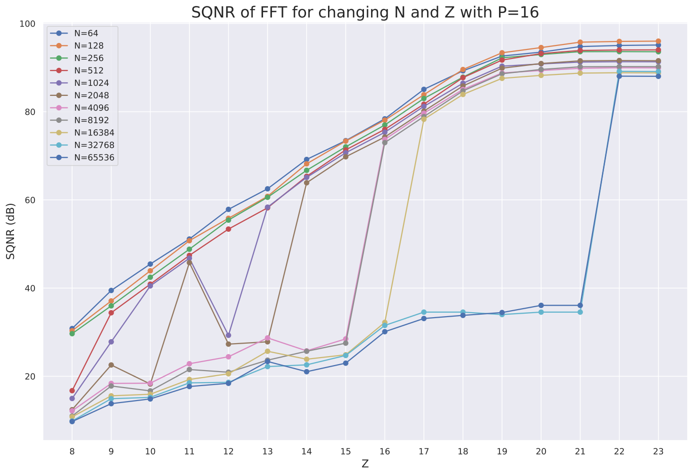
What do we see?

Higher bits → higher accuracy



TU/e

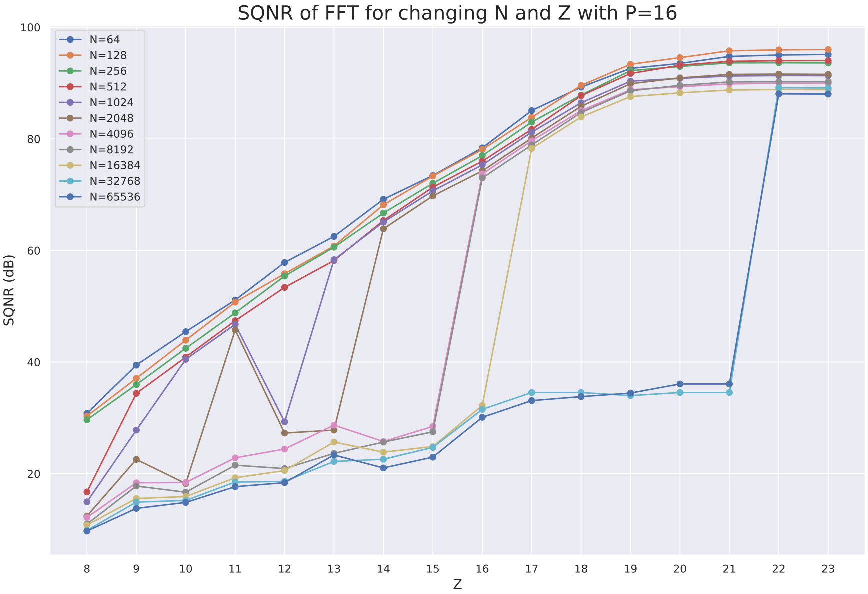
FFT only



SQNR of FFT for changing N and Z with P=16
SQNR (dB)
Z
N=64
N=128
N=256
N=512
N=1024
N=2048
N=4096
N=8192
N=16384
N=32768
N=65536
100
80
60
40
20
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

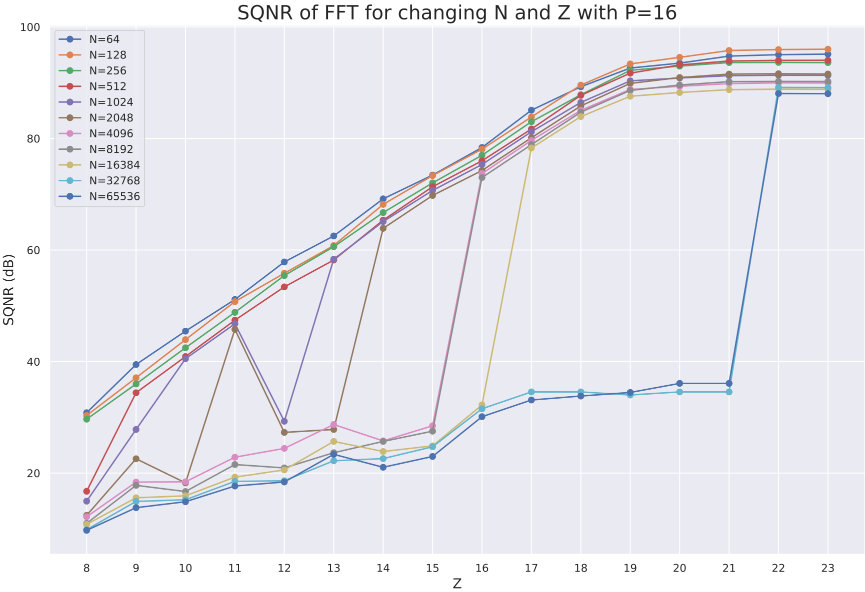
FFT

What do we see?



FFT

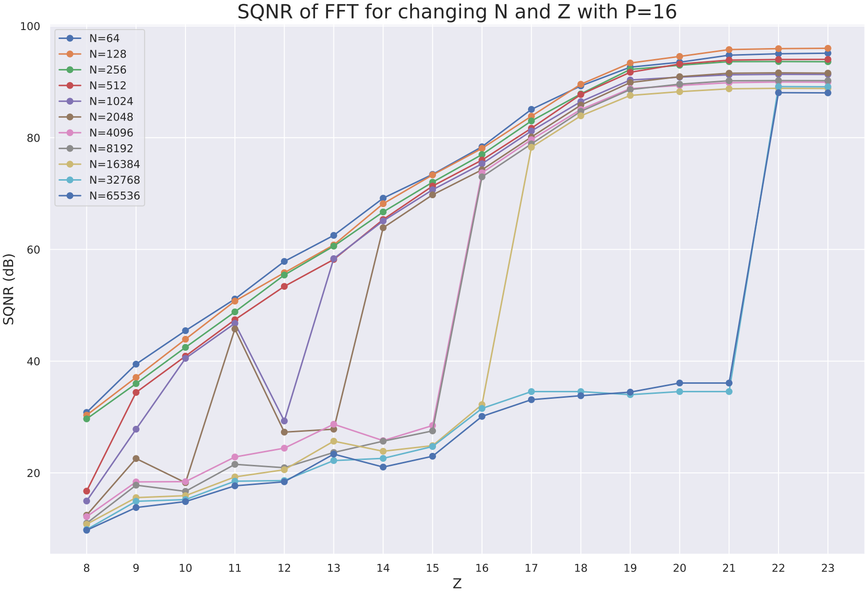
What do we see?
Higher bits → higher accuracy



FFT

What do we see?
Higher bits → higher accuracy

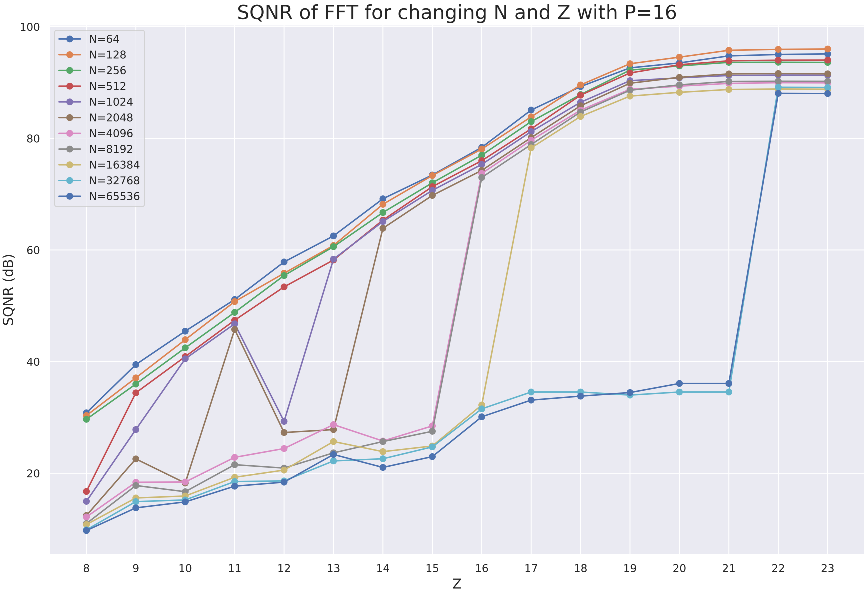
clear cut off point!
Why?



FFT

What do we see?
Higher bits $\rightarrow$
higher accuracy

clear cut off point!
Why?
possibly:
high N means:
very small twiddle



Future work

Hardware synthesis step from StaccatoLab (not done yet in library)

Future work

Hardware synthesis step from StaccatoLab (not done yet in library)

Make twiddle fabrication more accurate (more smooth FFT SQNR graph)

In Conclusion

In Conclusion

analysis presented:
Pulsar search -> signal with heavy frequency dispersion

In Conclusion

analysis presented:
Pulsar search -> signal with heavy frequency dispersion

cdmt: Frequency De-Dispersion through phase only filter

In Conclusion

analysis presented:
Pulsar search -> signal with heavy frequency dispersion

cdmt: Frequency De-Dispersion through phase only filter

State of the art on GPU: not real time and large intermediate storage
-> port to FPGA

In Conclusion

analysis presented:
Pulsar search -> signal with heavy frequency dispersion

cdmt: Frequency De-Dispersion through phase only filter

State of the art on GPU: not real time and large intermediate storage
-> port to FPGA

FFT redesigned with dataflow for FPGA

In Conclusion

analysis presented:
Pulsar search -> signal with heavy frequency dispersion

cdmt: Frequency De-Dispersion through phase only filter

State of the art on GPU: not real time and large intermediate storage
-> port to FPGA

FFT redesigned with dataflow for FPGA

library created to generate the scalable design

In Conclusion

analysis presented:
Pulsar search -> signal with heavy frequency dispersion

cdmt: Frequency De-Dispersion through phase only filter

State of the art on GPU: not real time and large intermediate storage
-> port to FPGA

FFT redesigned with dataflow for FPGA

library created to generate the scalable design

Design tested and verified in simulation

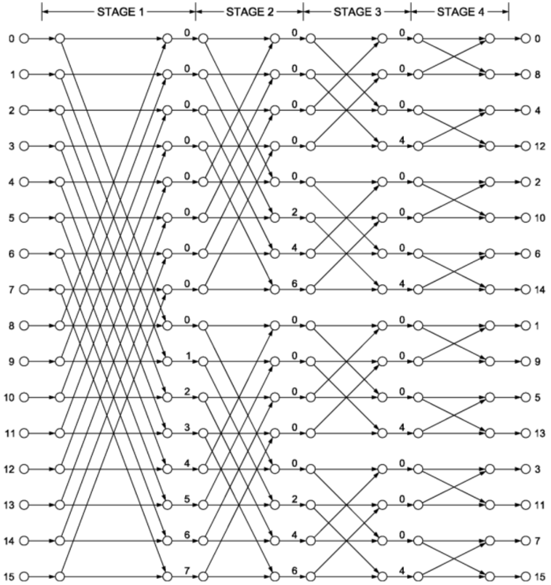
Questions?

Extra slides

The DFT

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}, k = 0, 1, ..., N-1 \tag{13}$$

$$W_N^{nk} = W_N^{\phi} = e^{\dfrac{-j \cdot 2\pi \cdot \phi}{N}} \tag{14}$$



Radix 2 $\rightarrow$ **Radix** 2^2

Split the twiddle (rotation on complex plane):

$$W_N^{nk} = W_N^{\phi} = e^{\dfrac{-j \cdot 2\pi \cdot \phi}{N}} \tag{15}$$

Radix 2 $\rightarrow$ Radix 2^2

Split the twiddle (rotation on complex plane):

$$W_N^{nk} = W_N^{\phi} = e^{\dfrac{-j \cdot 2\pi \cdot \phi}{N}} \tag{15}$$

into trivial and non trivial part:

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

Radix 2 $\rightarrow$ Radix 2^2

Split the twiddle (rotation on complex plane):

$$W_N^{nk} = W_N^{\phi} = e^{\dfrac{-j \cdot 2\pi \cdot \phi}{N}} \tag{15}$$

into trivial and non trivial part:

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

Trivial: $\{0, N/4, N/2, 3N/4\} \rightarrow \{0, \pi/2, \pi, 1.5\pi\}$
Trivial is already in R2, what about non trivial?

What is Fixed point arithmetic?

Simple example:
Floating point number one third: 0.33333333333333333333...

What is Fixed point arithmetic?

Simple example:
Floating point number one third: 0.33333333333333333333...
Multiply with large number and truncate

What is Fixed point arithmetic?

Simple example:
Floating point number one third: 0.33333333333333333333...
Multiply with large number and truncate
for example fit in 10 bits -> $\times 2^{10}$

What is Fixed point arithmetic?

Simple example:
Floating point number one third: 0.3333333333333333333...
Multiply with large number and truncate
for example fit in 10 bits -> $\times 2^{10}$

do math on integers $\rightarrow$ much easier on hardware!

What is Fixed point arithmetic?

Simple example:
Floating point number one third: 0.3333333333333333333...
Multiply with large number and truncate
for example fit in 10 bits -> $\times 2^{10}$

do math on integers $\rightarrow$ much easier on hardware!

shift result back with $\div 2^{10}$

What is Fixed point arithmetic?

Simple example:
Floating point number one third: 0.33333333333333333333...
Multiply with large number and truncate
for example fit in 10 bits -> $\times 2^{10}$

do math on integers $\rightarrow$ much easier on hardware!

shift result back with $\div 2^{10}$
make sure intermediate results keep fitting! $\rightarrow$ scale back intermediate

Scaling the bits

Input data: complex samples with 2 bytes (8 bits each)

Scaling the bits

Input data: complex samples with 2 bytes (8 bits each)
one for real, one for imaginary part

Scaling the bits

Input data: complex samples with 2 bytes (8 bits each)
one for real, one for imaginary part
fit into Z bits $\rightarrow$ shift number of 2^{Z-8}

Scaling the bits

Input data: complex samples with 2 bytes (8 bits each)
one for real, one for imaginary part
fit into Z bits $\rightarrow$ shift number of 2^{Z-8}

limiting scope of project, relative simple scaling:

Scaling the bits

Input data: complex samples with 2 bytes (8 bits each)
one for real, one for imaginary part
fit into Z bits $\rightarrow$ shift number of 2^{Z-8}

limiting scope of project, relative simple scaling:

- twiddle multipliers immediate scale back intermediate result

Scaling the bits

Input data: complex samples with 2 bytes (8 bits each)
one for real, one for imaginary part
fit into Z bits $\rightarrow$ shift number of 2^{Z-8}

limiting scope of project, relative simple scaling:

- twiddle multipliers immediate scale back intermediate result
- for butterfly addition $\rightarrow$ *orthogonal scaling*

the Inverse Fourier Transform and Orthogonal Scaling

making use of: $\frac{1}{N} = \frac{1}{\sqrt{N}} \cdot \frac{1}{\sqrt{N}}$

the Inverse Fourier Transform and Orthogonal Scaling

making use of: $\frac{1}{N} = \frac{1}{\sqrt{N}} \cdot \frac{1}{\sqrt{N}}$

rewrite the DFT and inverse definition:

$$X[k] = \frac{1}{\sqrt{N}} \cdot \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \qquad (17)$$

$$x_n = \frac{1}{\sqrt{N}} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn} \qquad (18)$$

the Inverse Fourier Transform and Orthogonal Scaling

making use of: $\frac{1}{N} = \frac{1}{\sqrt{N}} \cdot \frac{1}{\sqrt{N}}$

rewrite the DFT and inverse definition:

$$X[k] = \frac{1}{\sqrt{N}} \cdot \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \qquad (17)$$

$$x_n = \frac{1}{\sqrt{N}} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn} \qquad (18)$$

share the numerical size across both!

the Inverse Fourier Transform and Orthogonal Scaling

making use of: $\frac{1}{N} = \frac{1}{\sqrt{N}} \cdot \frac{1}{\sqrt{N}}$

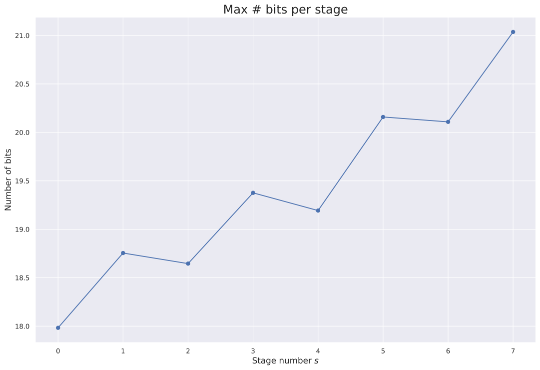
rewrite the DFT and inverse definition:

$$X[k] = \frac{1}{\sqrt{N}} \cdot \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{17}$$

$$x_n = \frac{1}{\sqrt{N}} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi}{N}kn} \tag{18}$$

share the numerical size across both!

Example with N=256 and Z=18

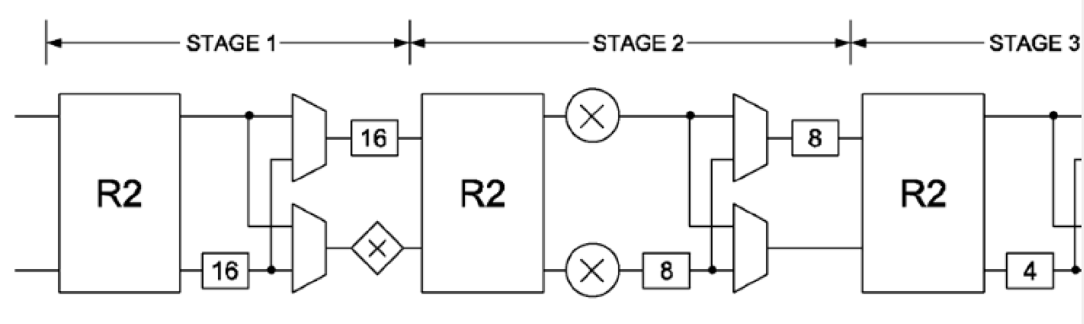


Connection Pattern

Depending on P: implemented as single vector in token

Connection Pattern

Depending on P: implemented as single vector in token
P=2 $\rightarrow$ simple straight



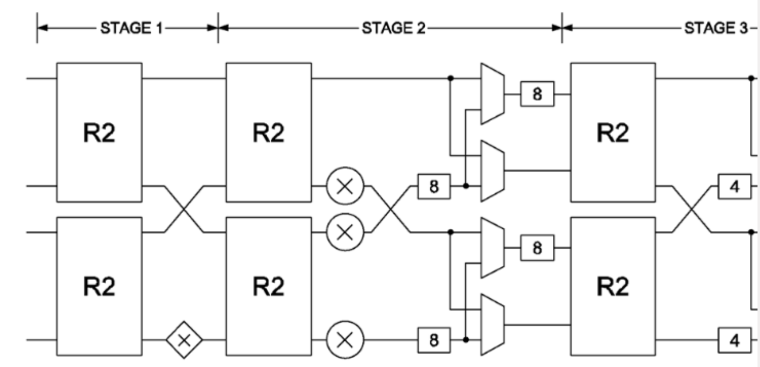
Connection Pattern (cont.)

Depending on P: implemented as single vector in token

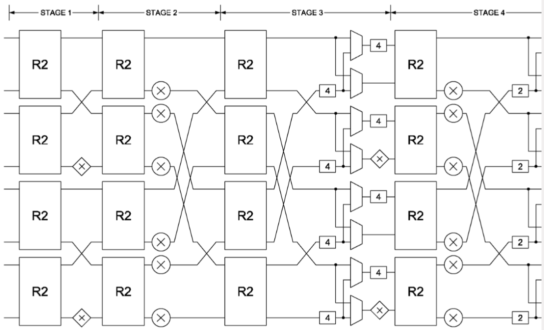
Connection Pattern (cont.)

Depending on P: implemented as single vector in token

P=4 $\rightarrow$ crossing pattern

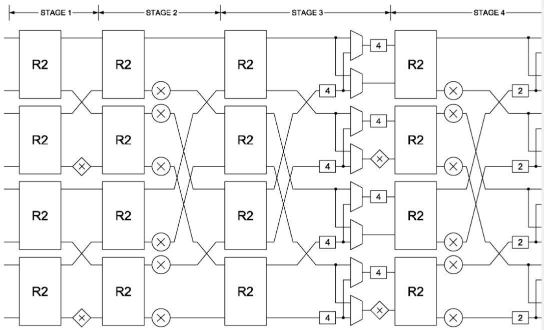


Connection Pattern (cont.)



Depending on P: implemented as single vector in token

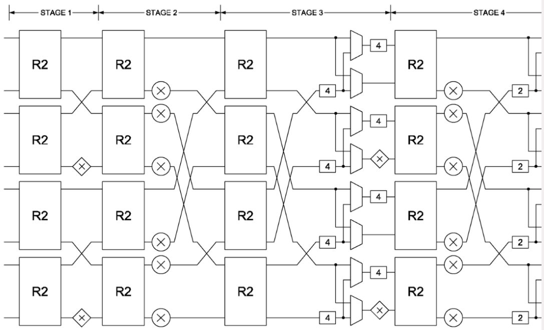
Connection Pattern (cont.)



Depending on P: implemented as single vector in token

P=8 $\rightarrow$ first repeat of pattern

Connection Pattern (cont.)



Depending on P: implemented as single vector in token

P=8 → first repeat of pattern

→ then stretched interleaving

Rotating factors

s=1, N=64, P=16
each row: non trivial rotating integers ϕ for one clock cycle

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

Rotating factors

s=1, N=64, P=16
each row: non trivial rotating integers ϕ for one clock cycle

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

P columns by $\frac{N}{P} = \frac{64}{16} = 4$ rows

How to generate?

Taking first row:

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

How to generate?

Taking first row:
4 groups of 4, consisting of:

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

How to generate?

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Taking first row:
4 groups of 4, consisting of:
$base \cdot \{0, 2, 1, 3\}$

How to generate?

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Taking first row:
4 groups of 4, consisting of:
$base \cdot \{0, 2, 1, 3\}$
with base multiples of $\frac{N}{P}$ $(= 4$ here)

How to generate?

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Taking first row:
4 groups of 4, consisting of:
$base \cdot \{0, 2, 1, 3\}$
with base multiples of $\frac{N}{P}$ $(= 4$ here)

in order of (again!) $\frac{N}{P} \cdot \{0, 2, 1, 3\}$

How to generate?

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Taking first row:
4 groups of 4, consisting of:
$base \cdot \{0, 2, 1, 3\}$
with base multiples of $\frac{N}{P}$ $(= 4$ here)

in order of (again!) $\frac{N}{P} \cdot \{0, 2, 1, 3\}$

each row increases base with 1

Hardware needed

Nodes:

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Hardware needed

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Nodes:

- $base_\alpha$

Realized as:

- CORDIC: hardware

Hardware needed

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

$$0 \cdot \frac{N}{P} \cdot \{0,2,1,3\}, 2 \cdot \frac{N}{P} \cdot \{0,2,1,3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0,2,1,3\}, 3 \cdot \frac{N}{P} \cdot \{0,2,1,3\}$$

Nodes:

- $base_\alpha$
- $R_{n\alpha}$ (with $n \in \{2,3\}$)

Realized as:

- CORDIC: hardware
- Rotation matrix using goniometric properties

Hardware needed

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

Nodes:

- $base_\alpha$
- $R_{n\alpha}$ (with $n \in \{2, 3\}$)
- $R_{N//P}$

Realized as:

- CORDIC: hardware
- Rotation matrix using goniometric properties
- Rotation matrix precomputed

Hardware needed

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

This is all done in single sample token mode!

$\rightarrow$ relative simple connection pattern

Dataflow implementation

after first stage only one of the four blocks

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

$$0 \cdot \frac{N}{P} \cdot \{0,2,1,3\}, 2 \cdot \frac{N}{P} \cdot \{0,2,1,3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0,2,1,3\}, 3 \cdot \frac{N}{P} \cdot \{0,2,1,3\}$$

Dataflow implementation

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

$$0 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 2 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}, 3 \cdot \frac{N}{P} \cdot \{0, 2, 1, 3\}$$

after first stage only one of the four blocks

with base increasing with $\left\lfloor \frac{s-1}{2} \right\rfloor$

Dataflow implementation

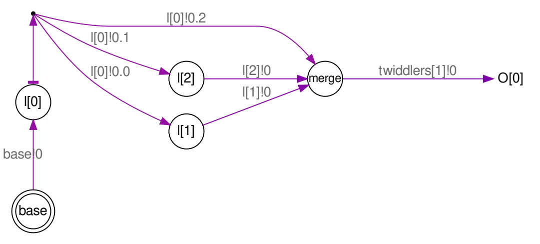
0	0	0	0	0	16	8	24	0	8	4	12	0	24	12	36
0	2	1	3	0	18	9	27	0	10	5	15	0	26	13	39
0	4	2	6	0	20	10	30	0	12	6	18	0	28	14	42
0	6	3	9	0	22	11	33	0	14	7	21	0	30	15	45

$$0 \cdot \frac{N}{P} \cdot \{0,2,1,3\}, 2 \cdot \frac{N}{P} \cdot \{0,2,1,3\},$$
$$1 \cdot \frac{N}{P} \cdot \{0,2,1,3\}, 3 \cdot \frac{N}{P} \cdot \{0,2,1,3\}$$

after first stage only one of the four blocks

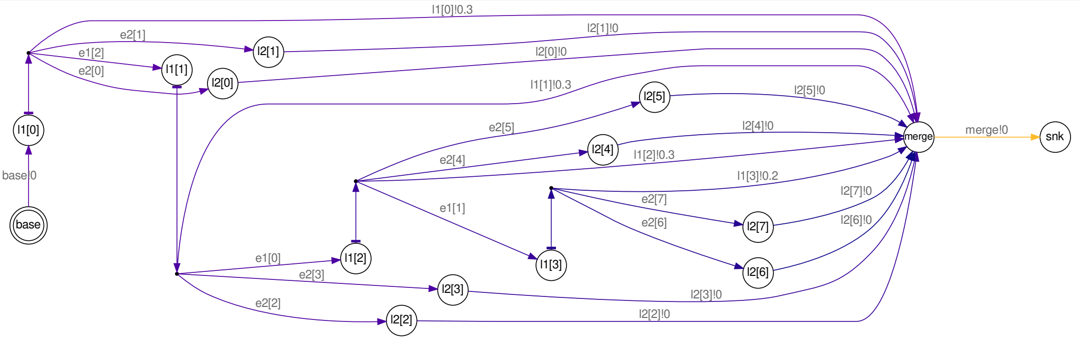
with base increasing with $\left\lfloor \frac{s-1}{2} \right\rfloor$

Graph form:



Dataflow implementation $s = 1$

Everything together produces this work of art:



CORDIC

$$
\begin{aligned}
x_{i+1} &= x_i - d_i \cdot 2^{-i} \cdot y_i \\
y_{i+1} &= y_i + d_i \cdot 2^{-i} \cdot x_i \\
\omega_{i+1} &= \omega_i - d_i \cdot \tanh(2^{-i}) \\
d_i &= -sign(\omega_i)
\end{aligned} \tag{19}
$$

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

Scaled Rotation Matrix

$$\begin{aligned}
\begin{bmatrix} x'_{2\alpha} \\ y'_{2\alpha} \end{bmatrix} &= \begin{bmatrix} \cos(2\alpha_0) & -\sin(2\alpha_0) \\ \sin(2\alpha_0) & \cos(2\alpha_0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x^2 - y^2 \\ 2 \cdot x \cdot y \end{bmatrix} \\
\begin{bmatrix} x'_{3\alpha} \\ y'_{3\alpha} \end{bmatrix} &= \begin{bmatrix} \cos(3\alpha_0) & -\sin(3\alpha_0) \\ \sin(3\alpha_0) & \cos(3\alpha_0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \cdot x^3 - 3 \cdot x \\ 3 \cdot y - 4 \cdot y^3 \end{bmatrix}
\end{aligned} \tag{20}$$

For the 3α rotation matrix

$$\left(\begin{bmatrix} x & -y \\ y & x \end{bmatrix} \cdot \begin{bmatrix} (x^2 - y^2) \gg Z \\ (2 \cdot x \cdot y) \gg Z \end{bmatrix} \right) \gg Z \tag{21}$$

Measure unit

Measured in Signal-to-quantization-noise ratio (SQNR):

$$SQNR(dB) = 10 \cdot \log_{10}\left(\frac{E\{\mid X_{ID} \mid^2\}}{E\{\mid X_Q - X_{ID} \mid^2\}}\right) \tag{22}$$

Measure unit

Measured in Signal-to-quantization-noise ratio (SQNR):

$$SQNR(dB) = 10 \cdot \log_{10}\left(\frac{E\{\mid X_{ID} \mid^2\}}{E\{\mid X_Q - X_{ID} \mid^2\}}\right) \tag{22}$$

Misleading name: not only quantization errors!
But Garrido uses it