

Porting DDfacet and KillMS onto AArch64

Etienne Renault- SiPearl

21st Jan 2026, CERN



Rhea1 and AArch64 ecosystem

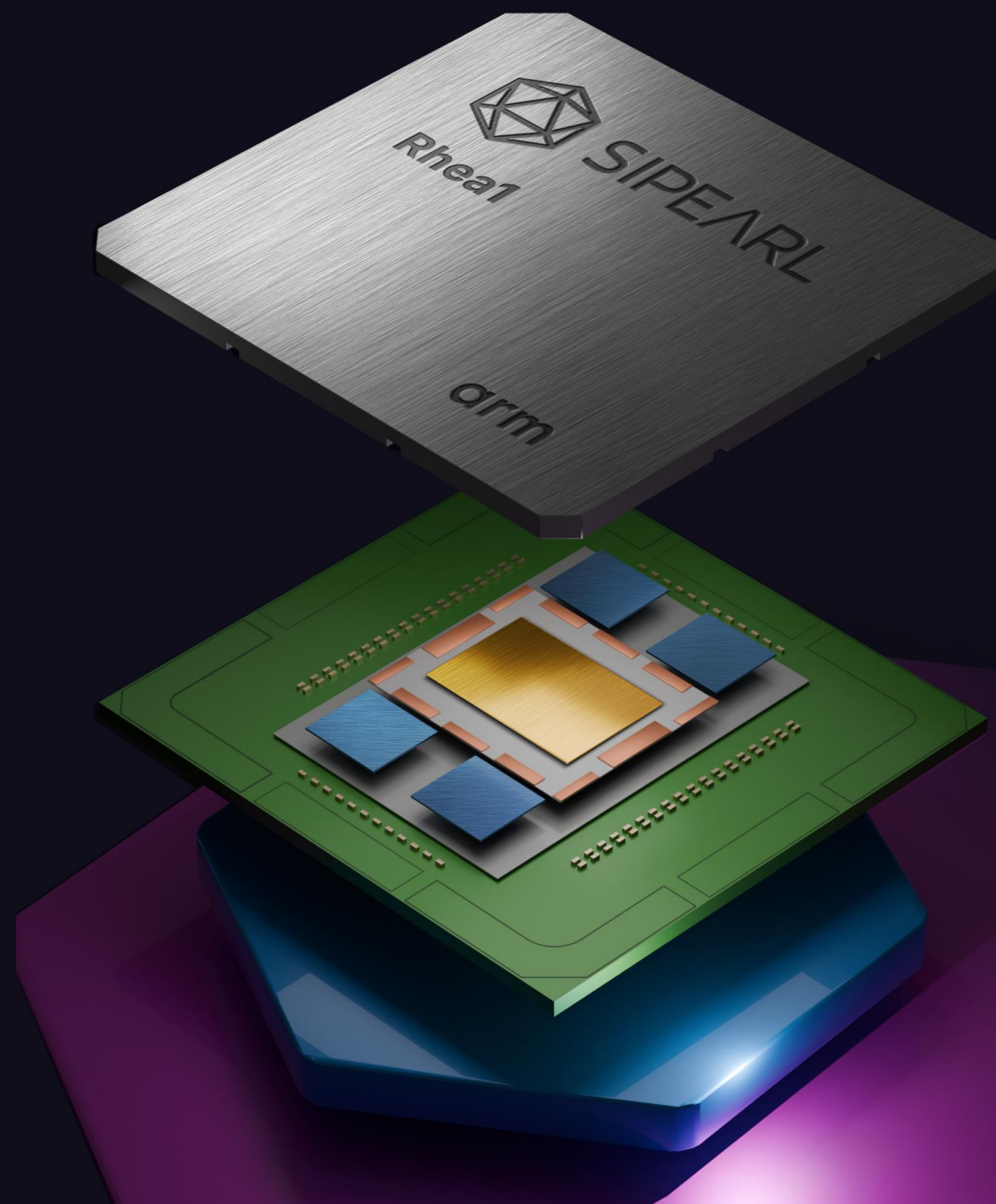
RHEA1

HPC and AI microprocessor

80 Arm® Neoverse V1 cores
with 2 x 256 SVE each

4 x HBM2e

4 x DDR5 interfaces



The Neoverse Series

N Serie

- General-purpose cloud computing & infrastructure.
- Balanced design: Focus on energy efficiency and predictable performance-per-watt.
- Cloud VMs, databases, general compute

V Serie

HPC (High-Performance Computing) & supercomputing.

Wide out-of-order pipelines, bigger cores, higher IPC.

Scientific workloads, AI/ML training, weather modeling, etc

E Serie

Edge computing, networking, and throughput-oriented tasks.

Smaller cores, often used in many-core designs

Packet processing, 5G infrastructure, content delivery networks.

Many Vector Extension for Neoverse V architecture

| AArch64 profile | NEON | SVE | SVE2 |
|------------------------|-------------|------------|-------------|
| Neoverse-N1 | ✓ | ✗ | ✗ |
| Neoverse-V1 | ✓ | ✓ | ✗ |
| Neoverse-V2 | ✓ | ✓ | ✓ |

NEON vectors: 128 bits
SVE vectors: from 128 to 2048 bits
SVE: Scalable Vector Extension

VLA

Vector Length Agnostic
programming model

Write once

Compile once

Vectorize more loops

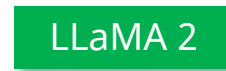
Illustration of Vector Length Agnostic Programming

```
void example01(int *restrict a, const int *b, const int *c, long N)
{
    long i = 0;
    //Pseudo code of vectorization
    auto predicate = /*...*/
    for (; i < N; i += 4)
    {
        <predicate[0]>? a[i]    = b[i]    + c[i];
        <predicate[1]>? a[i+1] = b[i+1] + c[i+1];
        <predicate[2]>? a[i+2] = b[i+2] + c[i+2];
        <predicate[3]>? a[i+3] = b[i+3] + c[i+3];
    } predicates update depending on i
}
```

one instruction

SW Stack for AArch64

Applications
Science & Models



Distributed High-level libraries



Networking
Storage



Communication



Data Storage



Shared-memory libraries
Runtime



arm COMPUTE LIBRARY



Low-level libraries



BSC



Online Data Intensive Solutions for Science in the Exabytes Era

Main Sipearl Contributions w.r.t. this Ecosystem

Toolchains

- LLVM contributions (vectorization, overall support incl. fortan)
 - + Maintainers rights
- AdaptiveCpp contributor to support SYCL performance and memory tiering support

Libraries

- Contributor to oneMath
 - UXL foundation member
- FFTW improvment
 - more details later in this presentation

Application

- Regular improvement of application based on extensive profiling, mostly at:
 - Memory level
 - Vectorisation level
- But also ability to improve on power thanks to specific PMU profing

Packaging

- Ability to provide latest software stacks (with latest sipearl's development) in the context of collaborative projects.



Case study: Porting ddf-pipelines on AArch64

*Low-Frequency Array (LOFAR) radio
telescope*

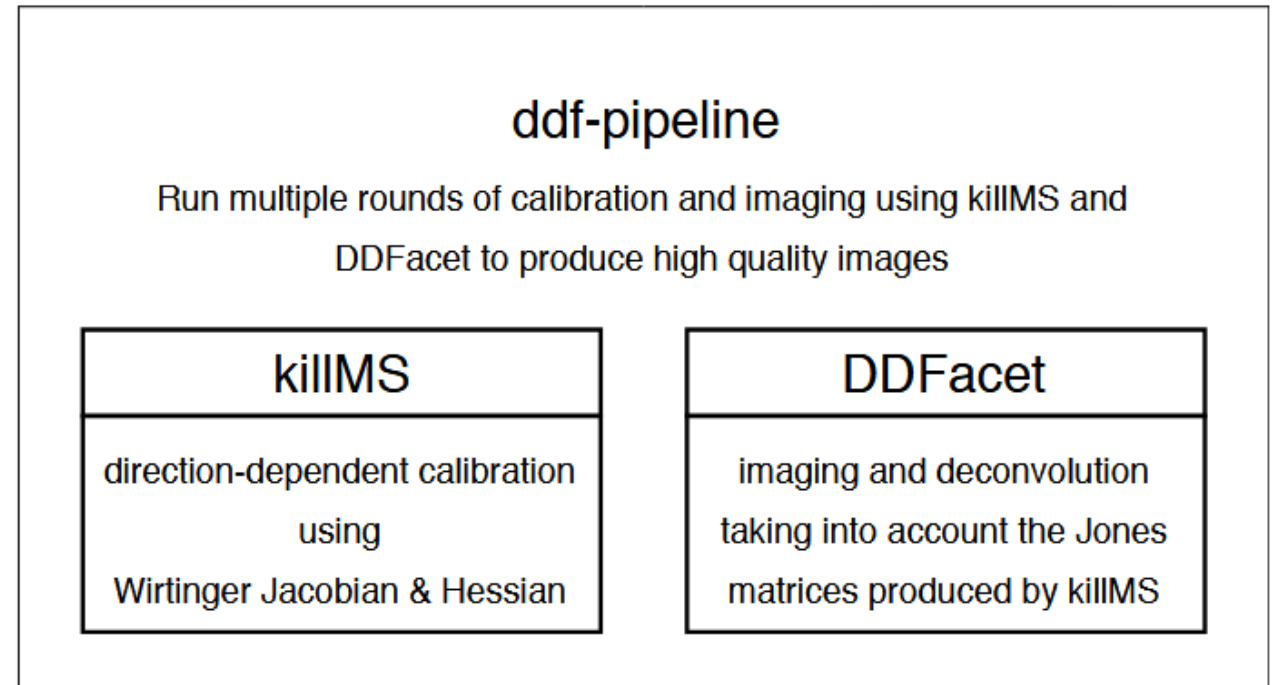
DDF-pipeline Software Stack

DDF-pipeline performs
Calibration/Residual (CS&RS)array
Direction-(In)Dependent calib. & imaging.

Software Stack

- Python (front/back end)
- C++ (backend: numpy, pyfftw)
- Allows distribution over cluster nodes
- Allows resuming of interrupted jobs

**Significant compute and memory
throughput requirements.**



DDF pipeline Singularity

- DDF pipeline is distributed through a containerization environment
 - Popular in HPC realm since its creation (2015)
 - Compliant with OCI (Open Container Initiative) such as docker and podman
 - Rootless
 - Seamless access to Accelerator and file systems
 - Designed to handle complex application and data intensive workloads
 - Typically the case here since LOFAR generates
 - terrabytes of data per hour after, and
 - requires extensive post-processing for imaging and calibration.
- Massive amount of dependencies
 - Linux packages
 - Python package
 - Git to be built from scratch

DDF pipeline Singularity

DDF pipeline traditionally deployed for X86

*Some channels, e.g. I4DS has only X86 builds (e.g "ska" packages
ska-sdp-datamodels ska-sdp-func ska-sdp-func-python)*

Based on a Debian bullseye (version'11 – 2024)

Current Debian is version13 – 2025

Challenge#1: Porting

*Provide an up-to-date AArch64 compliant
– with latest development that support SVE –
on a software that has never been ported on
aarch64*

Challenge#2: Profile and Optimize

*Provide a insights on performance to drive future
optimisations*

Challenge #1

Challenge #1: Porting

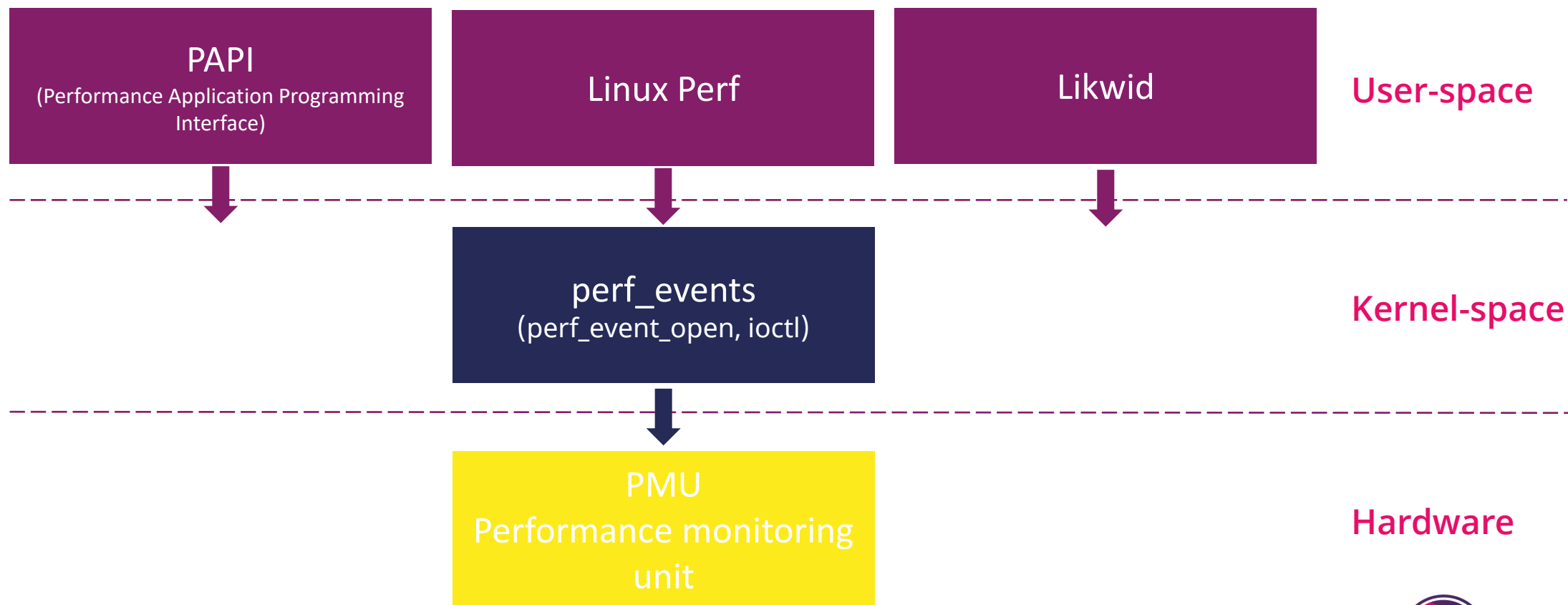
Imaging: (ported and tested on Aarch64)
Calibration: KillIMS (WIP)

- Singularity image was updated to use latest Debian + associated Aarch64 packages.
- Python dependency conflicts.
 - Rebuild not AArch64 packages (not build by official pip repo)
 - Stabilising a common testing environment on Aarch64 (conda/pip).
 - Integrate Numpy (OpenBLAS fallback) optimizations for SVE
 - OpenBlas not built by default with this support
 - *Still Fujitsu effort to be accepted upstream [\[1\]](#)*
- LibFFTW SVE support
 - upstreamed by SiPearl
 - not packaged by default
 - see following slides
- Still work in progress
 - Support of efficient math
 - Sleef / Libm : not packaged , WIP

Challenge #2: Profiling DDFacet

How to access Performance Monitoring Unit (PMU)?

- Performance information are usually accessed through hardware level monitoring counters.
- We generally use the perf_events interface to access PMU:



Perf Example

| Name | Typical Use | Typical Scale | Languages | Metrics |
|------|--|----------------|----------------------------|-------------------------------|
| Perf | Quick detailed profile of a single process | Single process | Anything, usually compiled | Wallclock time, hardware perf |

List of pre-defined events (to be used in `-e` or `-M`):

```
branch-misses [Hardware event]
bus-cycles [Hardware event]
cache-misses [Hardware event]
cache-references [Hardware event]
cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
stalled-cycles-backend OR idle-cycles-backend [Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]
alignment-faults [Software event]
bpf-output [Software event]
cgroup-switches [Software event]
context-switches OR cs [Software event]
cpu-clock [Software event]
cpu-migrations OR migrations [Software event]
dummy [Software event]
emulation-faults [Software event]
major-faults [Software event]
minor-faults [Software event]
page-faults OR faults [Software event]
task-clock [Software event]
duration_time [Tool event]
user_time [Tool event]
system_time [Tool event]
```

```
general:
cnt_cycles
  [Constant frequency cycles. Unit: armv8_pmu3_0]
cpu_cycles
  [Cycle. Unit: armv8_pmu3_0]
l1d_cache:
l1d_cache
  [Level 1 data cache access. Unit: armv8_pmu3_0]
l1d_cache_inval
  [L1D cache invalidate]
l1d_cache_lmiss_rd
  [Level 1 data cache long-latency read miss. Unit: armv8_pmu3_0]
l1d_cache_rd
  [L1D cache access, read]
l1d_cache_refill
  [Level 1 data cache refill. Unit: armv8_pmu3_0]
l1d_cache_refill_inner
  [L1D cache refill, inner]
l1d_cache_refill_outer
  [L1D cache refill, outer]
l1d_cache_refill_rd
  [L1D cache refill, read]
l1d_cache_refill_wr
  [L1D cache refill, write]
l1d_cache_wb
  [Attributable Level 1 data cache write-back. Unit: armv8_pmu3_0]
l1d_cache_wb_clean
  [L1D cache Write-Back, cleaning and coherency]
l1d_cache_wb_victim
  [L1D cache Write-Back, victim]
l1d_cache_wr
  [L1D cache access, write]
```

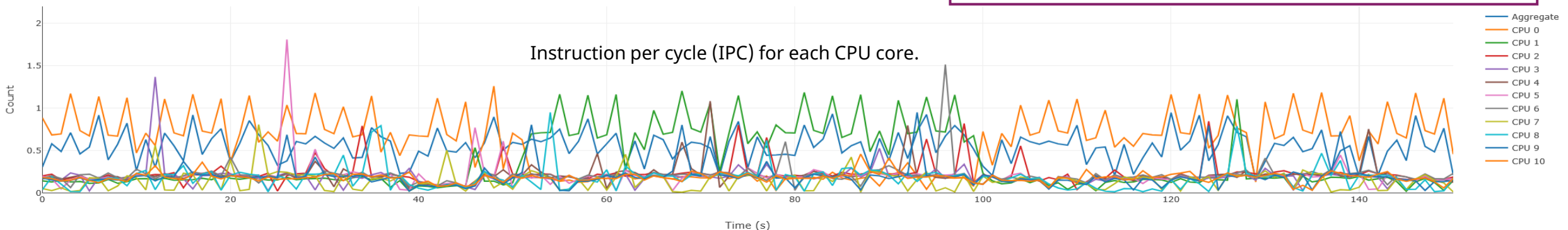
Example: DDFacet performance analysis using APerf

- Per thread analysis shows also the different compute behavior, current analysis shows highly serial code, with most CPU cores idles and $\frac{1}{2}$ cores running.
- High code parallelization is difficult due to the nature of the FFTW3 which involves many CPU instructions, and some parallelizable deconv operations,
- Testing over a small dataset and setting output image generation to 500 pix (`--Image-NPix=500`), we observe 8% L2 and 56% L3 miss rates, with an execution time of 196,8s on a Neoverse-N1 platform (Ampere altra).

Unclear view of parameter space exploration and representativeness of

python DDF.py

```
--Data-MS mslist.txt
--Output-Name out
--Output-Mode Clean
--Deconv-Mode SSD2
--Deconv-MaxMajorIter 3
--Weight-ColName None
--Mask-Auto 1
--Output-RestoringBeam 15
--SSD2-PolyFreqOrder 2
--Freq-NBand 2
--Freq-NDegridBand 5
--Image-NPix=500
--Parallel-NCPU 9
```



MAQAO

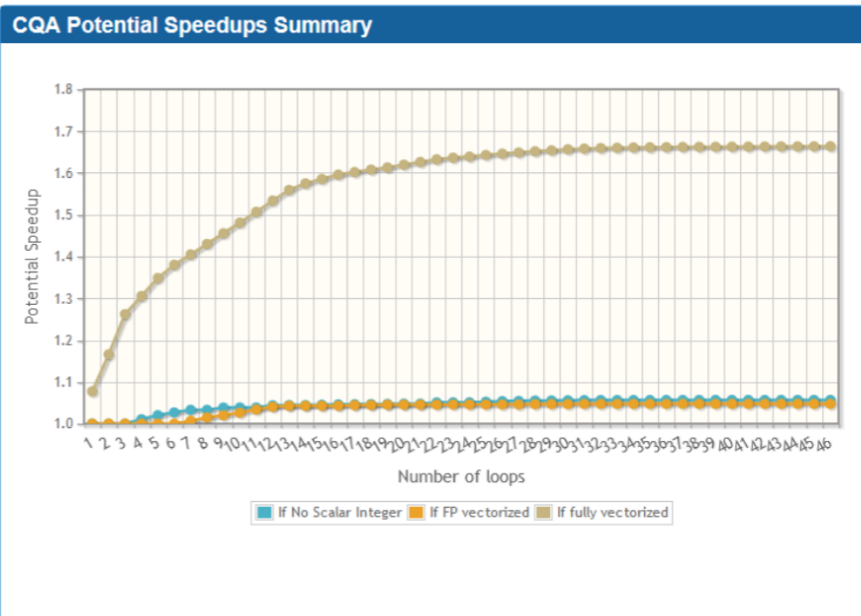
| Name | Typical Use | Typical Scale | Languages | Metrics |
|-------|---|---------------|-------------------|-----------------------------------|
| MAQAO | Analysis of the profile an application | Node level | C,C++, Fortran | Reports similar to vtune (x86) |

MAQAO Global Application Functions Loops Topology

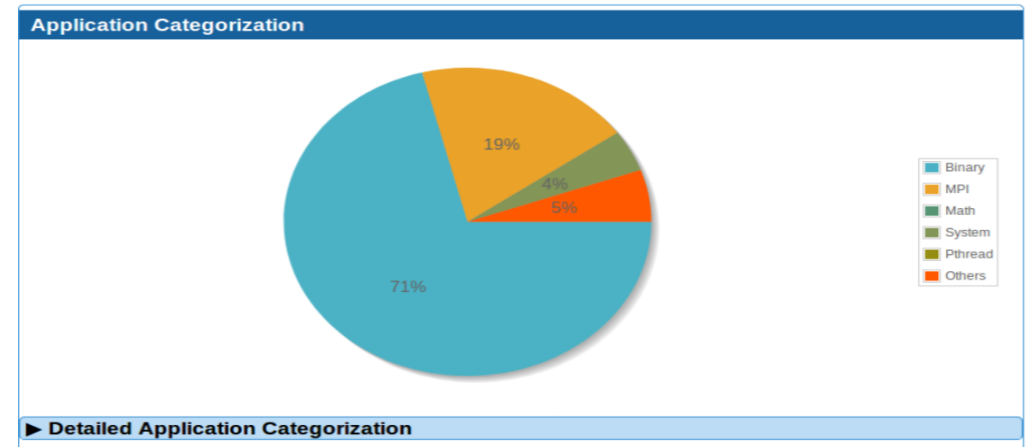
bt-mz.A.x - 2021-02-19 15:21:14 - MAQAO 2.12.6

Help is available by moving the cursor above any ? symbol or by checking [MAQAO website](#).

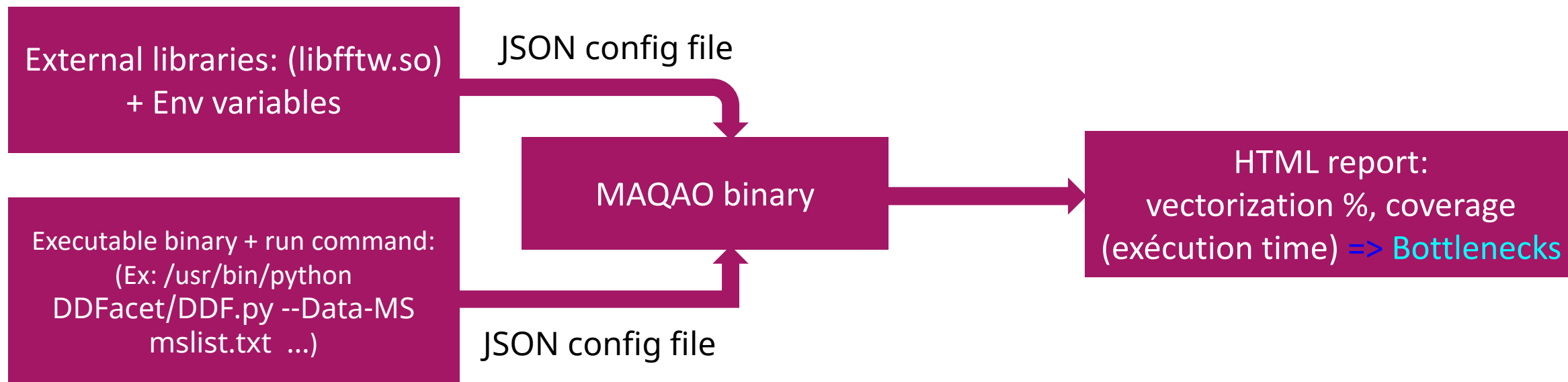
| Global Metrics | |
|--|--|
| Total Time (s) | 10.3 |
| Profiled Time (s) | 10.3 |
| Time in loops (%) | 53.57 |
| Time in innermost loops (%) | 51.76 |
| Time in user code (%) | 97.24 |
| Compilation Options | bt-mz.A.x -march=(target) is missing -funroll-loops is missing |
| Perfect Flow Complexity | 1.04 |
| Array Access Efficiency (%) | 96.44 |
| Perfect OpenMP + MPI + Pthread | 1.01 |
| Perfect OpenMP + MPI + Pthread + Perfect Load Distribution | 1.03 |
| No Scalar Integer | Potential Speedup: 1.06 Nb Loops to get 80%: 8 |
| FP Vectorised | Potential Speedup: 1.05 Nb Loops to get 80%: 6 |
| Fully Vectorised | Potential Speedup: 1.66 Nb Loops to get 80%: 12 |



| Source Location | Source Function | Level | Coverage run_0 (%) | Time w.r.t. Wall Time run_0 (s) | Vectorization Ratio (%) |
|--|--|-----------|--------------------|---------------------------------|-------------------------|
| ecrad - radiation_ifs_rrtm.F90:564-566 | radiation_ifs_rrtm_gas_optics | Innermost | 21.9 | 8.76 | 0 |
| ecrad - rrtm_gas_optical_depth.F90:184-184 | rrtm_gas_optical_depth | Innermost | 3.38 | 1.35 | 25 |
| ecrad - radiation_ifs_rrtm.F90:484-484 | radiation_ifs_rrtm_gas_optics | Innermost | 3.31 | 1.32 | 85.71 |
| ecrad - radiation_adding_ica_sw.F90:111-116 | radiation_adding_ica_sw_adding_ica_sw | Innermost | 2.78 | 1.11 | 95.45 |
| ecrad - radiation_two_stream.F90:478-542 [...] | radiation_two_stream_calc_reflectance_transmittance_sw | Single | 2.78 | 1.11 | 100 |
| ecrad - radiation_mcica_sw.F90:270-270 | radiation_mcica_sw_solver_mcica_sw | Innermost | 2.44 | 0.98 | 90.91 |
| ecrad - radiation_ifs_rrtm.F90:704-704 | radiation_ifs_rrtm_plank_function_atmos | Innermost | 2.21 | 0.89 | 83.33 |
| ecrad - radiation_mcica_lw.F90:294-294 | radiation_mcica_lw_solver_mcica_lw | Innermost | 1.95 | 0.78 | 88.89 |
| ecrad - random_numbers_mix.F90:185-191 | random_numbers_mix_initialize_random_numbers | Innermost | 1.59 | 0.64 | 0 |
| ecrad - radiation_two_stream.F90:376-394 | radiation_two_stream_calc_no_scattering_transmittance_lw | Single | 1.55 | 0.62 | 0 |
| ecrad - radiation_adding_ica_sw.F90:137-144 | radiation_adding_ica_sw_adding_ica_sw | Single | 1.49 | 0.59 | 94.74 |

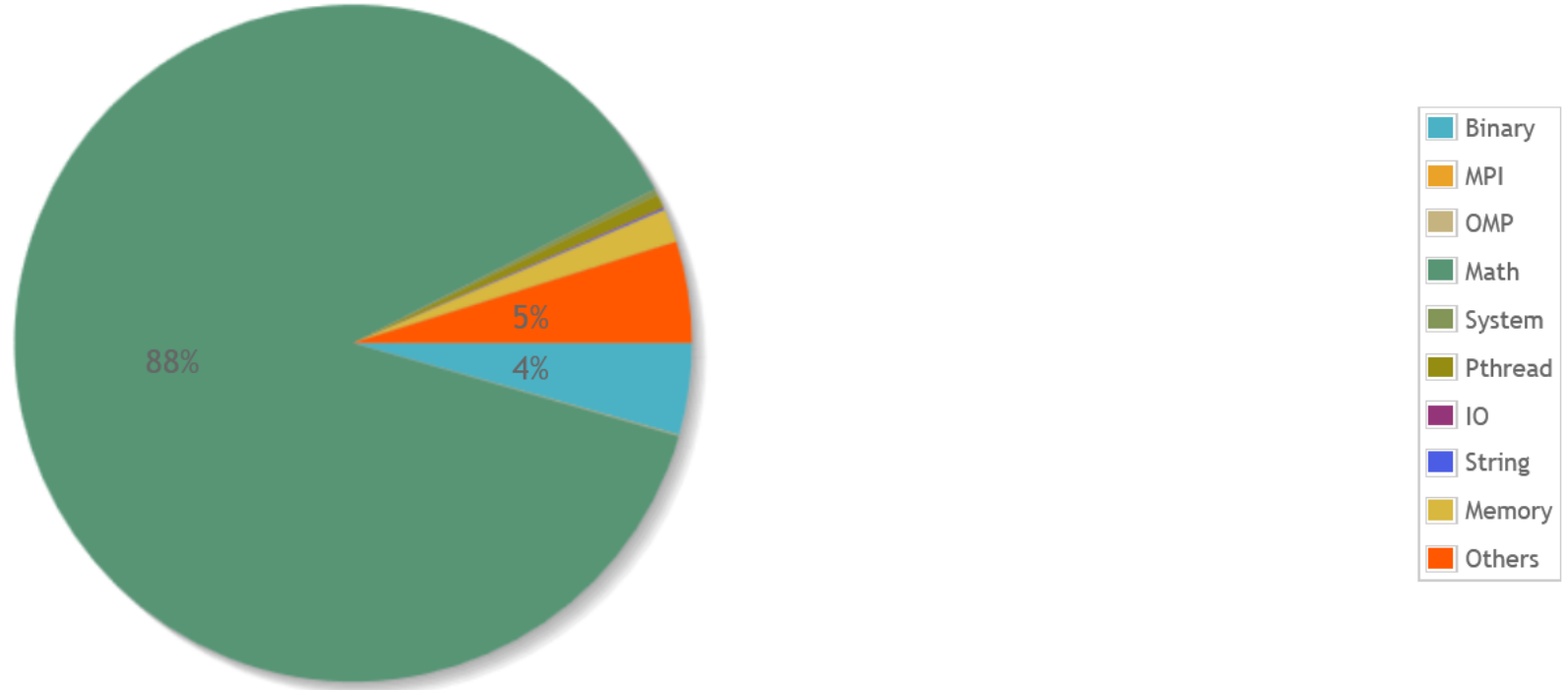


Example: Using MAQAO with a python written workload



LoFar Example: DDFacet performance analysis using MAQAO

- Running a short test case on the DDFacet algorithm, performance has been profiled and analysed:
- Maqao report reveals several key points about the workload performance.
- Python compliant



| ID | Time(s) | Binary(%) | MPI(%) | OMP(%) | Math(%) | System(%) | Pthread(%) | IO(%) | String(%) | Memory(%) | Others(%) |
|-------------------|---------|-----------|--------|--------|---------|-----------|------------|-------|-----------|-----------|-----------|
| ▼ run_0 | 20.14 | 4.39 | 0.02 | 0.04 | 88.04 | 0.31 | 0.65 | 0.05 | 0.07 | 1.60 | 4.84 |
| ▼ Node ampere01 | 20.14 | 4.39 | 0.02 | 0.04 | 88.04 | 0.31 | 0.65 | 0.05 | 0.07 | 1.60 | 4.84 |
| ▼ Process 4061022 | 20.14 | 4.39 | 0.02 | 0.04 | 88.04 | 0.31 | 0.65 | 0.05 | 0.07 | 1.60 | 4.84 |

LoFar Example: DDFacet performance analysis using MAQAO

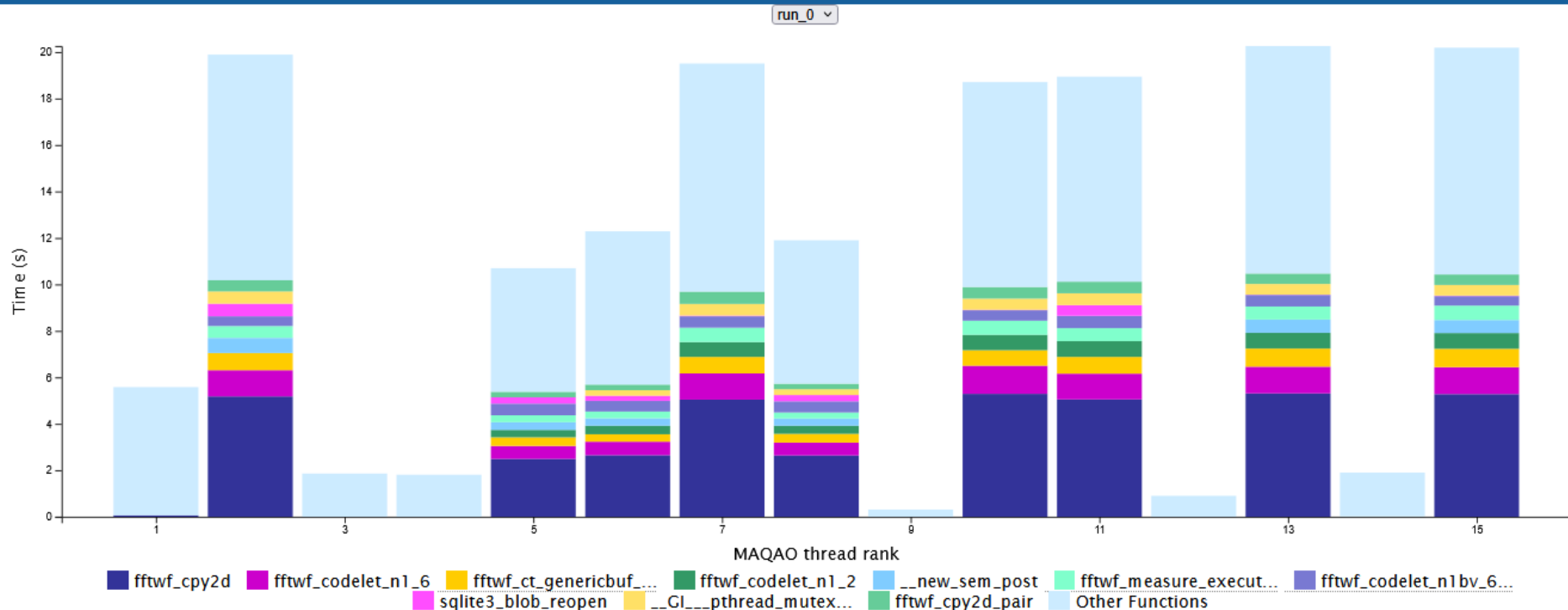
- Several key functions can be discovered when using MAQAO similarly to when using Linux Perf.
- The results show the percentage of the execution time of each function. As well as the percentage of vectorization of each function when necessary Libraries are linked.

| Name | Module | Coverage run_0 (%) | Coverage Excluding Loops run_0 (%) | Max Inclusive Time Over Threads run_0 (s) | Max Exclusive Time Over Threads run_0 (s) | Inclusive Time w.r.t. Wall Time run_0 (s) | Exclusive Time w.r.t. Wall Time run_0 (s) | Nb Threads run_0 | Deviation (coverage) run_0 | Deviation (walltime) run_0 | Categories run_0 | Compilatio Options |
|--------------------------------|----------------------------|-----------------------|---|--|--|--|--|------------------------|----------------------------------|----------------------------------|---------------------|-----------------------|
| ○ fftwf_cpy2d | libfftw3 f.s o.3.5.8 | 24.29 | 24.29 | 5.24 | 5.24 | 4.81 | 4.81 | 10 | 7.79 | 1.77 | Math (%): 100.00 | |
| ○ fftwf_codelet_n1_6 | libfftw3 f.s o.3.5.8 | 5.54 | 0 | 1.23 | 0.00 | 1.10 | 0.00 | 9 | 0.32 | 0.27 | Math (%): 100.00 | |
| ○ fftwf_ct_genericbuf_register | libfftw3 f.s o.3.5.8 | 3.38 | 3.38 | 0.78 | 0.78 | 0.67 | 0.67 | 10 | 1.12 | 0.25 | Math (%): 100.00 | |
| ○ fftwf_codelet_n1_2 | libfftw3 f.s o.3.5.8 | 3.14 | 0 | 0.72 | 0.00 | 0.62 | 0.00 | 9 | 0.21 | 0.16 | Math (%): 100.00 | |
| ○ fftwf_measure_execution_time | libfftw3 f.s o.3.5.8 | 2.84 | 2.84 | 0.65 | 0.65 | 0.56 | 0.56 | 9 | 0.25 | 0.14 | Math (%): 100.00 | |

Example: DDFacet performance analysis using MAQAO

- Per thread analysis shows also the different compute intensive operations:
- Libfftw functions seem to be the bottleneck of this workload, though more thorough analysis needs to be taken for different number of threads, size of dataset, parallelism, backend libraries ... ect
- **Disclaimer:** Work in progress/small test dataset.

Detailed Function Times



- LOFAR imaging workloads(ex. DDFacet) include many FFT operations from the Libfftw library.
- FFTs are algorithms used to reduce the computational complexity of doing a full Discrete Fourier Transform to $O(N\log(N))$:
 - Leveraging the many redundancies in the constant matrix
- The FFTW3 library is commonly used
 - Proprietary libraries often implements the FFTW3 interface (Intel MKL, Arm PL)

Libfftw implementation provides an optimized implementation to reduce the computational complexity on the vectorization of FFTW codelets on SIMP units:

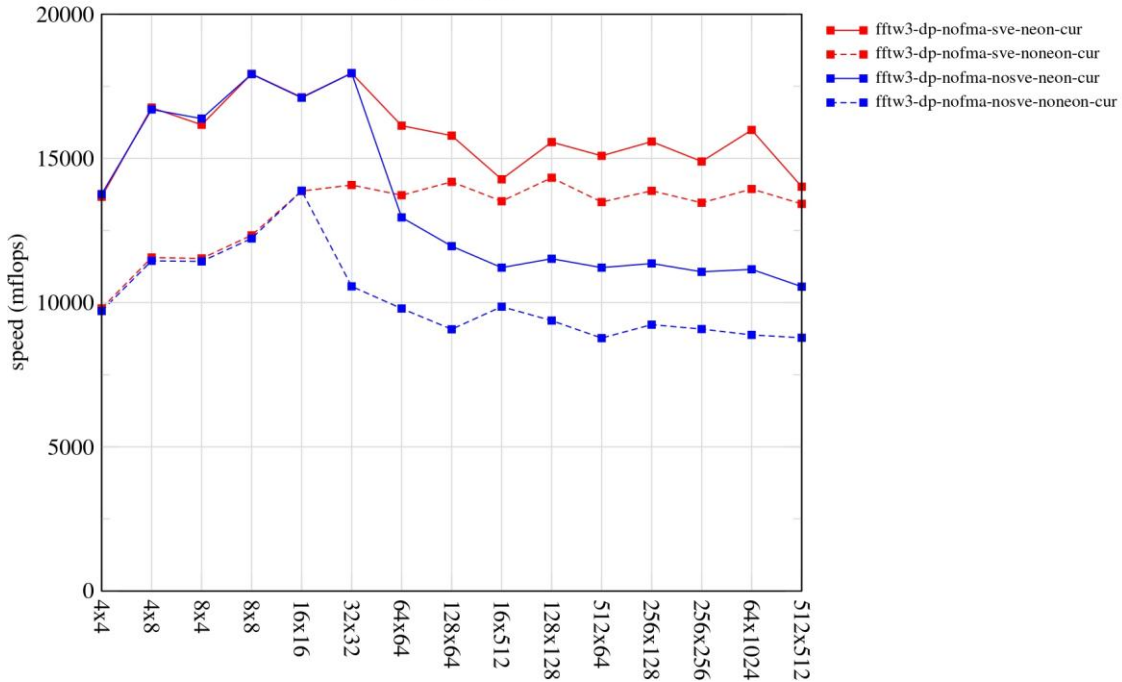
- **AVX2/AVX512** for x85
- **NEON/SVE** for Aarch64

- FFTW3 adds the vector-level parallelism in the loop implementing “ N_1 transforms of size N_2 ” and “ N_2 transforms of size N_1 ”
 - **Vector length applied against N_1 and N_2 respectively**
- Arm SVE is reasonably well adapted to implementing FFTs “the FFTW3 way”.
- FFTW3 Has been optimized for Aarch64’s SVE and evaluated on Graviton3 CPUs.
- Graviton3 are based on ARM neoverse V1 similar to Sipearl’s Rhea1.

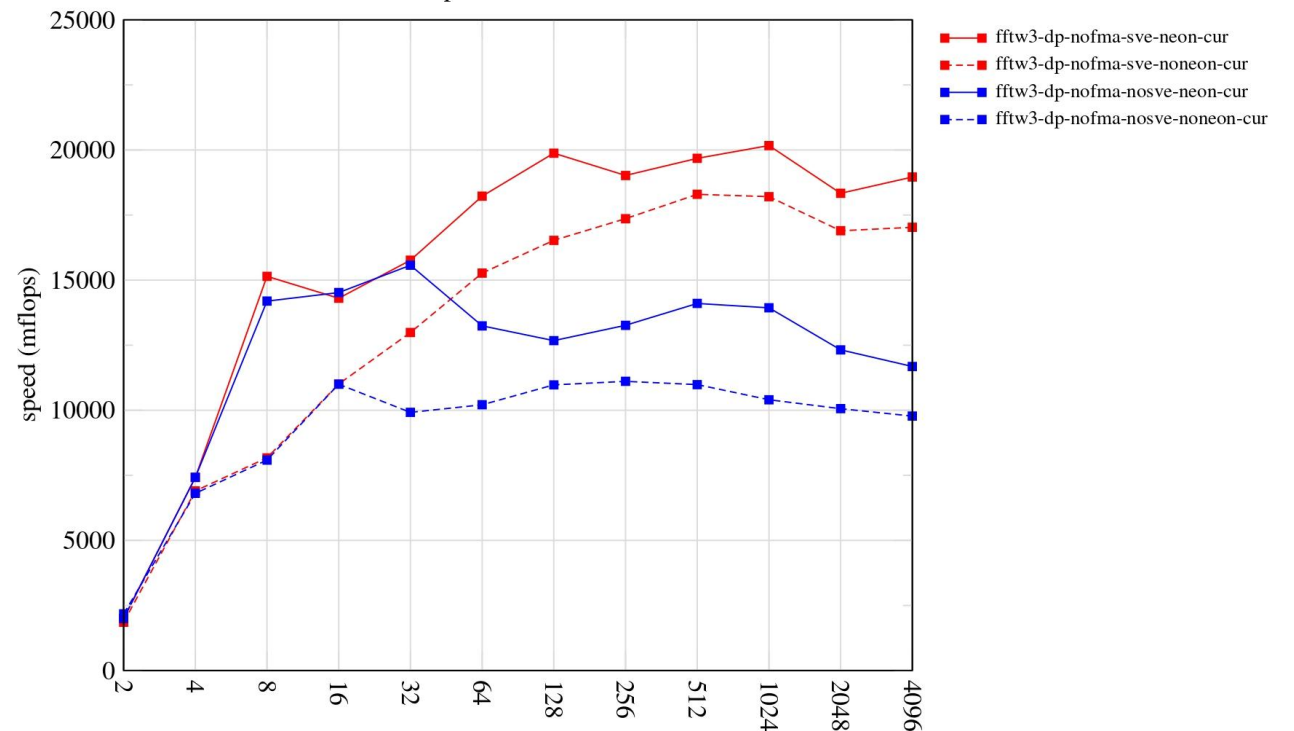
FFTW3 & SIMD support

- FFTW3 is self-tuning, so for a given transform, the executed code may vary!!!.
- The performance analysis of SVE optimizations can be shown for 1D and 3D dimensions for complex numbers as follows: **Red is for SVE, Blue is for NEON.** “Higher is better”

double-precision complex, 2d transforms
powers of two



double-precision complex, 1d transforms
powers of two



Conclusion and Future Work


A need for code-owner to rely on the latest software stack

- Porting of radio imaging pipeline for radio imaging on Aarch64:
 - Karabo-pipeline porting for: Orchestration (Karabo) ; Simulation (OSKAR) and Calibration (Quartical - WIP)
 - Only X86 tests so far – lot of python libs with deps.
 - Challenge to find a stable environment for testing (conda/pip environment)
- Ensure compatibility with accelerator
- Characterizing algorithm's bottlenecks and profiling hotspots
 - Optimise CPU related part / ensure efficient SW stack with any accelerator

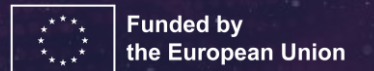


ODISSEE

Online Data Intensive Solutions
for Science in the Exabytes Era

 Odissee-project.eu

 [ODISSEE Project](#)



Funded by
the European Union