Recommended Developments Necessary for Applying (W)Asp Deconvolution Algorithms to ngVLA

G. Hsieh, S. Bhatnagar, R. Hiriart, M. Pokorny

March 2022

Abstract

This memo characterizes the computational load of deconvolution algorithms, focusing on the new AspClean and Wide-band AspClean (WAsp), in order to estimate the number of computing resources required by ngVLA and evaluate the use of GPUs for deconvolution. Our performance profiling results show the hotspots and memory consumption of the deconvolution algorithms and recommend ways to utilize GPU for speedup for ngVLA. Our preliminary implementation of using the GPU-based FFT (`cuFFT`) in the hotspot function of AspClean shows a 2x runtime improvement, which is a lower-limit on the speedup. This demonstrates that utilizing GPU for the hotspot functions of deconvolution algorithms is a worthwhile line of work to pursue with data transfer between CPU-GPU memories factored in. Our analysis also shows that (W)Asp has improved memory performance which suggests an advanced memory system may not be required for applying (W)Asp to ngVLA.

1 Introduction

The ngVLA is currently in the Conceptual Design stage with the purpose of, "down-select the implementation options as far as possible and to confirm the detailed scope of the project, including the cost-driving and performance-driving requirements and key design features". The main goal of this work is to help define the ngVLA data processing requirements, which can then be used as input to define detailed requirements for the relevant systems (e.g. ngCASA in this case). Together with architecture changes and implementation plans for the required capabilities, we can process and analyze ngVLA datasets and images.

So far our work in the area has been concentrated in elaborating the scalability and computing sizing requirements, specifically characterizing the computational load of gridding in order to estimate the number of computing resources required by ngVLA (see [2]) and evaluate the use of GPUs for gridding. The memo summarizes the work that characterizes the cost of another subject, deconvolution, as well as evaluates the use of GPU for deconvolution. The work focuses on the new (W)Asp deconvolution algorithms because they have shown improved imaging performance than the widely used MS-Clean and MS-MFS algorithm for datasets from EVLA and ALMA ([3]).

The outline of the memo is as follows. Section 2 describes the computational characterizations, including imaging performance, hotspots and memory consumption, of the (W)Asp deconvolution

algorithms and their comparisons to MS-Clean/MS-MFS as a baseline reference. Section 3.1 summarizes the findings from the performance characterization results, identifies hotspots of the (W)Asp algorithms, and proposes ways to utilize GPU for runtime improvement. Section 4.2 describes our proof-of-concept study that uses the GPU-based FFT, `cuFFT`, to speed up the hotspot function in AspClean. It shows the line of work is worth pursuing in ngCASA with data transfer between CPU-GPU memories factored in. Section 5 summarizes the development plans for applying the (W)Asp deconvolutions to ngVLA.

2 Performance Profiling

The analysis tool used for this study is *Intel® VTune™ Profiler* ([1])). It can locate hotspots, the most time-consuming parts of the code and visualize hot code paths and time spent in each function and with its callees with Flame Graph. It can also identify the most significant hardware issues and pinpoint memory-access-related issues such as cache misses and high-bandwidth problems. To analyze the computing cost of narrow-band and wide-band deconvolution, we use a simulation of a jet and lobe like structure to showcase a realistic situation, namely thin jets as well as a mix of compact and extended structure. The dataset has 5 channels, going from 1 GHz to 2 GHz for the VLA D-config. The following sections show the imaging performance and cost of computing (runtime and memory consumption) of AspClean and WAsp on the "jet" dataset as well as the comparison with MS-Clean and MS-MFS as a baseline reference.

2.1 Imaging Performance

For the narrow-band imaging, Figure 1 shows that the current MS-Clean solution (middle) cannot get the spectral index correct for the long edges of the jet part, and the AspClean (right) has the spectral index much closer to the truth. Figure 2 shows the residual images for the MS-Clean (left) and AspClean (right). The Asp-Clean residual image is more noise-like than MS-Clean.

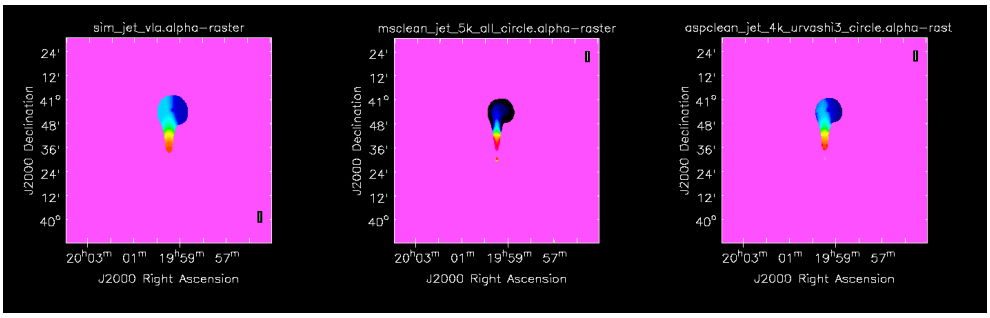


Figure 1: Spectral index comparison between the truth (left), MS-Clean (middle), and AspClean (right) on the jet dataset.

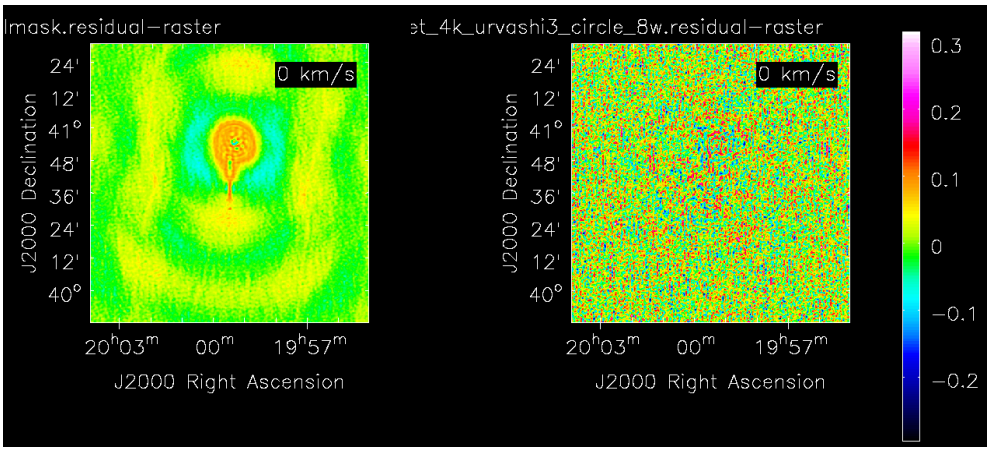


Figure 2: Residual image comparison between the MS-Clean (left) and AspClean (right) on the jet dataset.

Similarly, for the wide-band imaging Figure 3 shows that the current MS-MFS solution (middle) cannot get the spectral index correct for the long edges of the jet part, and the WAsp (right) has the spectral index much closer to the truth. Figure 4 shows the first, second and third-order Taylor coefficient residual images for the MS-MFS (top) and WAsp (bottom). The WAsp residual images are more noise-like than MS-MFS.

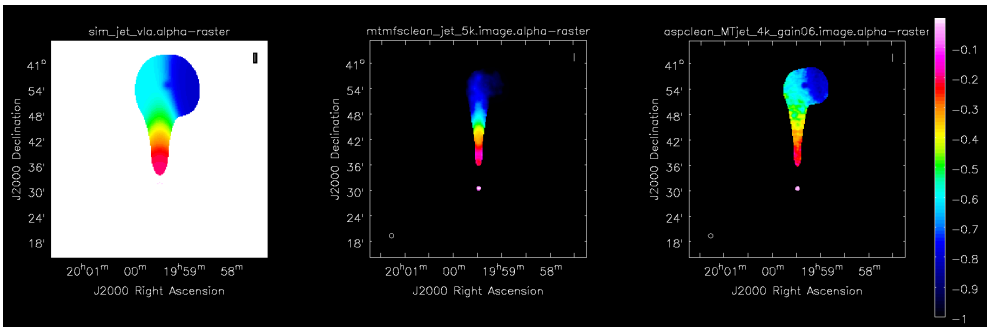


Figure 3: Spectral index comparison between the truth (left), MS-MFS (middle), and WAsp (right) on the jet dataset.

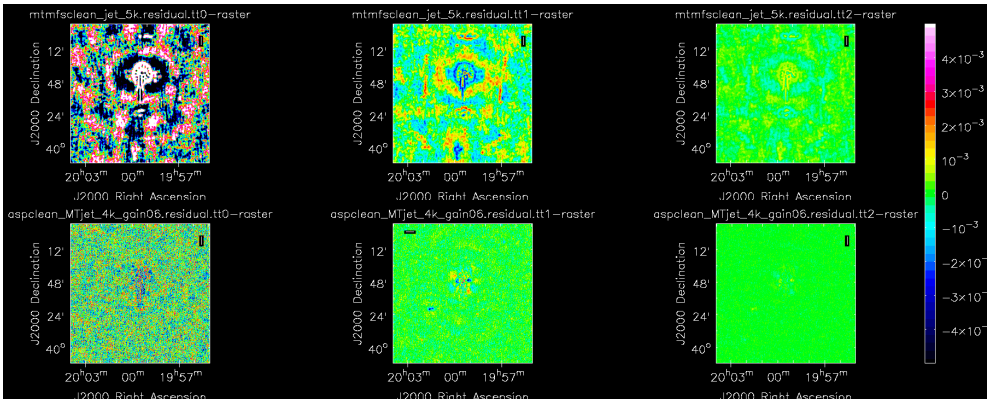


Figure 4: Residual image comparison between the MS-MFS (top) and WAsp (bottom) on the jet dataset.

2.2 Hotspots

2.2.1 AspClean

Figure 5 shows the screen shot of the performance profiling result of AspClean from the *Profiler*. It summarizes the overall performance of AspClean, including the runtime, hotspots, memory consumption, etc. Clicking the Hotspots icon in the window further gives detailed hotspots analysis (Figure 6) . The hotspots analysis in Figure 6 shows that the most time-consuming function of AspClean is `casacore::objcopy`. The `casacore::objcopy` is called by `casacore::FFTServer::flip` in the BFGS optimization function, `objfunc_alglib`, which is called by the `getActiveSetAspen` function that selects an initial Asp component and optimizes it ([3]).

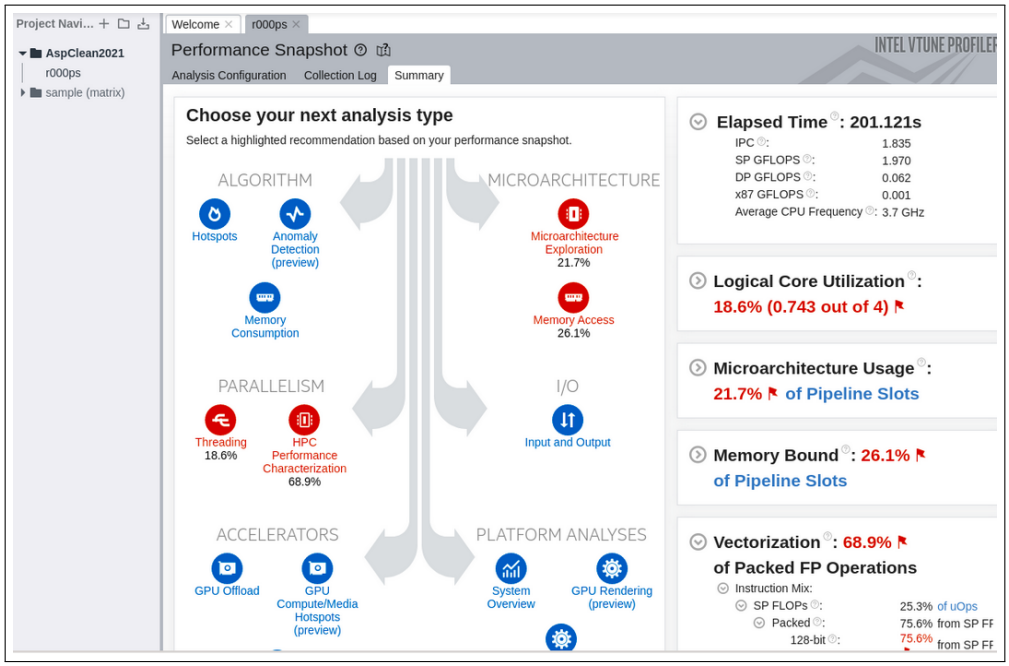
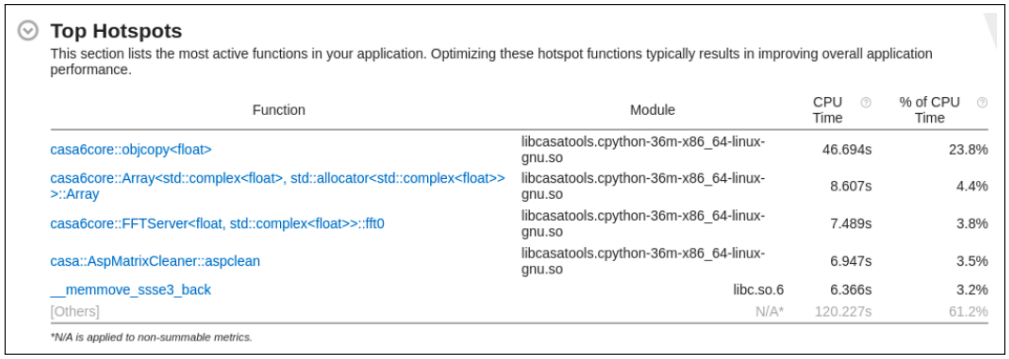


Figure 5: Performance snapshot of AspClean



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
casa6core::objcopy<float>	libcasatools.cpython-36m-x86_64-linux-gnu.so	46.694s	23.8%
casa6core::Array<std::complex<float>, std::allocator<std::complex<float>>>::Array	libcasatools.cpython-36m-x86_64-linux-gnu.so	8.607s	4.4%
casa6core::FFTServer<float, std::complex<float>>::fft0	libcasatools.cpython-36m-x86_64-linux-gnu.so	7.489s	3.8%
casa::AspMatrixCleaner::aspclean	libcasatools.cpython-36m-x86_64-linux-gnu.so	6.947s	3.5%
__memmove_ssse3_back	libc.so.6	6.366s	3.2%
[Others]	N/A*	120.227s	61.2%

*N/A is applied to non-summable metrics.

Figure 6: Top hotspots of AspClean

Figure 7 shows the detailed breakdown of the AspClean hotspots. It shows 68% of AspClean runtime is spent on the `aspclean` function. 83% of the `aspclean` runtime is spent on `getActiveSetAspen` and 7% of the `aspclean` runtime is spent on both `casacore::FFTServer::fft0` and `casacore::FFTServer::flip` that are outside of the `getActiveSetAspen` function. The `getActiveSetAspen`

function is the main function of AspClean that dynamically determines optimal scales at every minor iteration using the BFGS optimization, so not surprisingly it takes the majority of the runtime. 80% of the `getActiveSetAspen` runtime is spent on the BFGS optimization (i.e. `alglib::minlbfgsoptimize`) and 10% of `getActiveSetAspen` is on both `casacore::FFTServer::fft0` and `casacore::FFTServer::flip`. Almost all of the `alglib::minlbfgsoptimize` runtime is spent on `casa::objfunc_alglib`. Furthermore, 83% of the `casa::objfunc_alglib` runtime is spent on `casa::FFTServer::flip` (about 40%) and on `casa::FFTServer::fft0` (about 22% each).

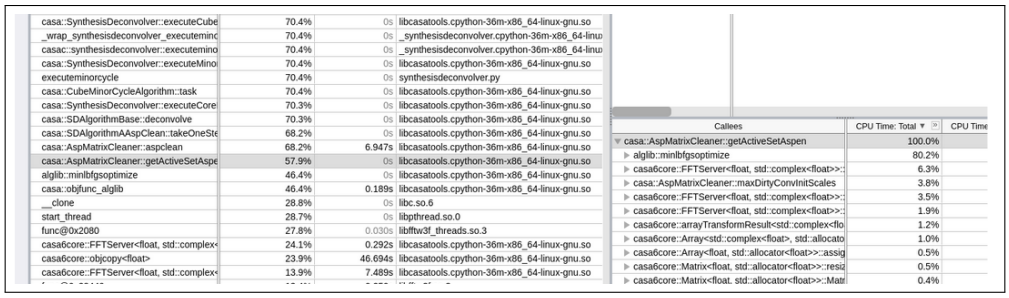


Figure 7: Top-down performance analysis of the AspClean hotspots

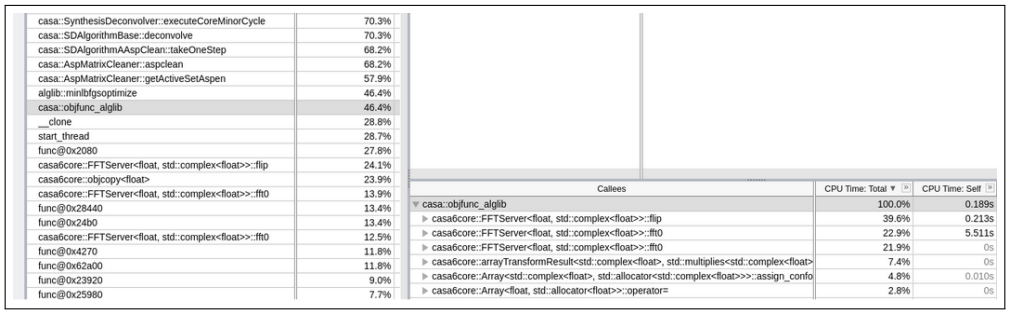


Figure 8: Top-down performance analysis of the `casacore::objcopy` function

Based on these results, the total runtime spent on `casacore::FFTServer::fft0` and `casacore::FFTServer::flip` is calculated as

$$0.83 * 0.8 * 0.83 + 0.83 * 0.1 + 0.07 = 0.703$$

The total runtime spent on matrix operations is,

$$0.1 + 0.83 * 0.1 + 0.83 * 0.8 * 0.17 = 0.297$$

In summary, about 70% of the `aspclean` runtime is spent on both `casacore::FFTServer::fft0` and `casacore::FFTServer::flip` and about 30% of `aspclean` is spent on matrix operations that are not localized in one function. For the BFGS optimization, matrix calculation (17%) is considered cheap compared to FFT (83%).

It is worth noting that `casacore::objcopy` is the most time-consuming function in AspClean, but not WAsp even though both algorithms call the same BFGS optimization function. This is possibly because AspClean has extra `casacore::FFTServer::fft0` and `casacore::FFTServer::flip` every minor iteration for $PSF * optimal\_scale$ to update the residual. The `casacore::FFTServer::flip` calls `casacore::objcopy`. Flipping is required because the `FFTServer` member functions all assume that the origin of the transform is at the center of the array, i.e. at $[nx/2, ny/2, ...]$, where the indexing begins at zero. Because the fftpack software assumes the origin of the transform is at the first element,

i.e. $[0, 0, ...]$, the `FFTServer` class flips the data in the array around to compensate.

2.2.2 MS-Clean

The hotspots analysis in the Figure 9 and Figure 10 show that the majority of the MS-Clean runtime is spent on the thread cloning and the `findMaxAbsMask` functions in `MatrixCleaner::clean`. The `findMaxAbsMask` function finds the peak value from an image matrix with mask applied. Unlike AspClean, `casacore::FFTServer` functions are not hotspots in MS-Clean mostly because it does not require the BFGS optimization for dynamically optimizing scales.

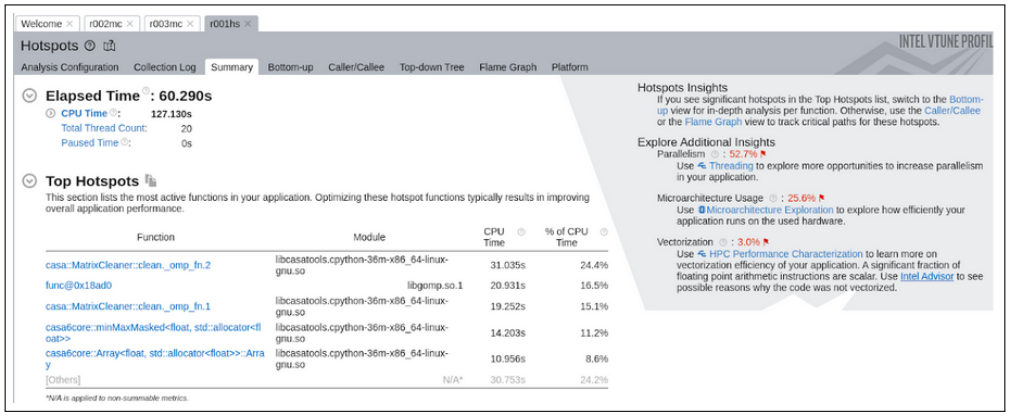


Figure 9: Top hotspots of MS-Clean

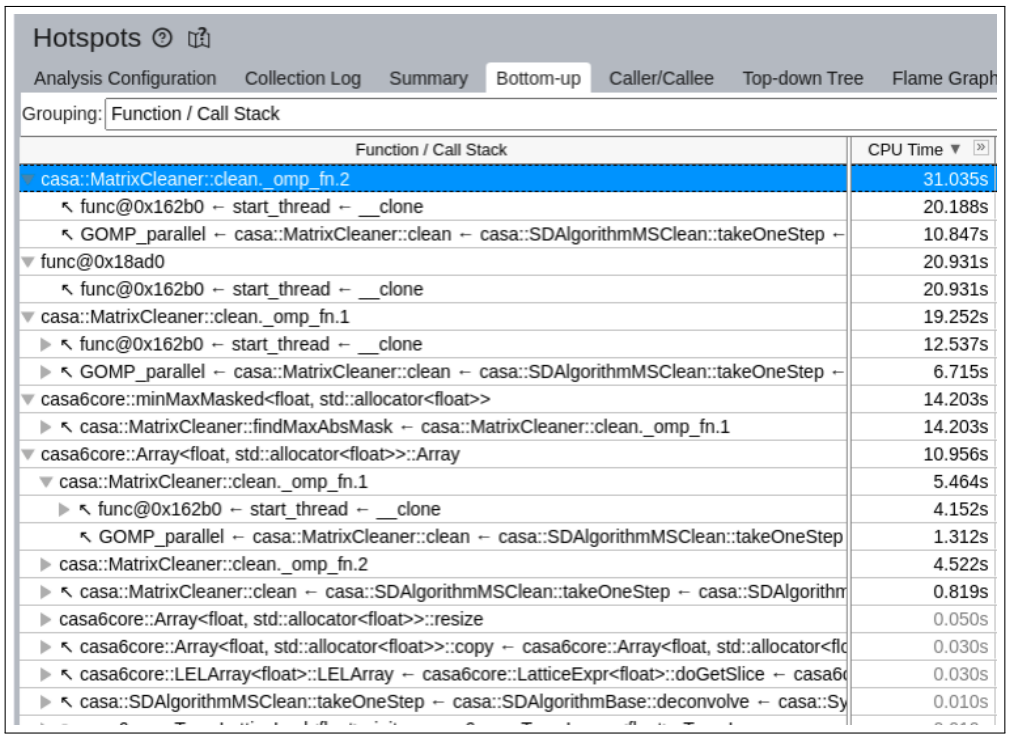


Figure 10: Top-down performance analysis of the MS-Clean hotspots

2.2.3 WAsp

The hotspots analysis in Figure 11 shows that WAsp does not have the time-consuming `casacore::objcopy` function. More time is spent on matrix operations (12%) than `fft0` (9%) which implies WAsp may benefit from performance improvement by moving matrix operations to GPU.

Furthermore, the detailed analysis (Figure 12 and Figure 13) show that 63% of WAsp runtime is spent on the `mtaspclean` function. 62% of the `mtaspclean` runtime is spent on `computeHessianPeak`. 41% of `computeHessianPeak` is on `casacore::FFTServer::fft0` and the rest 59% is on matrix operations. This is because `computeHessianPeak` does the following matrix operations:

- Calculate the convolutions of PSF and the optimal scale ($O(ntaylor^2 * 4)$ of `fft0` and matrix operation)
- Compute Hessian and inverse of Hessian

Also, compared with MS-MFS, `computeHessianPeak` in WAsp has

- an additional `findMaxAbs` function because of the nature of the Asp algorithm
- additional `matA_p[scale].resize(tgip, false); invMatA_p[scale].resize(tgip, false);` for filling up the two matrices.

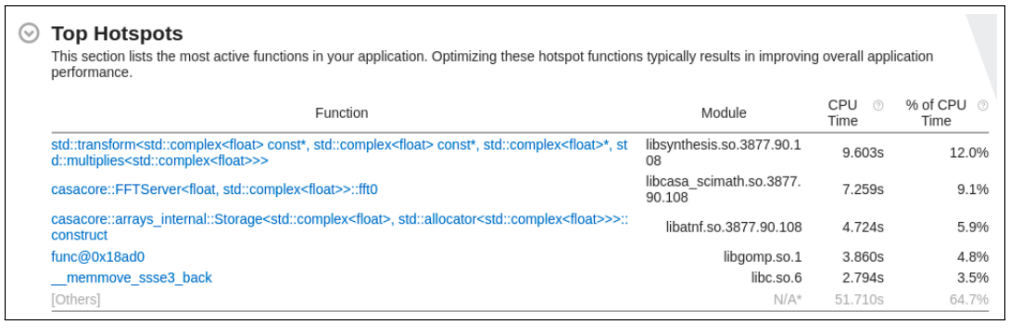
In WAsp, the `getActiveSetAspen` function has $O(numberofInitialScales^2)$ `casacore::FFTServer::flip` and `fft0` and operations to find peak from residual image convolved with all initial scales. The analysis shows that 17% of `mtaspclean` is spent on `getActiveSetAspen`. 25% of `getActiveSetAspen` is on `minlbfgsoptimize` (80% of `minlbfgsoptimize` is also on `FFTServer::flip` and `fft0`), 41% of `getActiveSetAspen` is on `casacore::FFTServer::flip` and `fft0`, and the rest is on matrix operation. That is, for WAsp totally 73% of the `getActiveSetAspen` runtime is spent on `casacore::FFTServer::flip` and `casacore::FFTServer::fft0`, and the rest is spent on matrix operations. Besides, 8% of `mtaspclean` is spent on `updateModelAndRHS` (mostly matrix operations), 3.3% of `mtaspclean` is on `casacore::FFTServer::fft0`, and 2.9% of `mtaspclean` is on `solveMatrixEqn` (mostly matrix operations). Thus, for WAsp the overall runtime of FFT is calculated as,

$$0.17 * 0.73 + 0.62 * 0.41 = 0.3783$$

, and the overall runtime of matrix operations is calculated as,

$$0.17 * 0.27 + 0.62 * 0.59 = 0.4117$$

That is, 38% of WAsp is on `casacore::FFTServer::fft0` and `flip`, and 41% of WAsp is on matrix operations that are more localized in `computeHessianPeak`. GPU may be used to speed up the matrix operations in `computeHessianPeak` in this case.



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
std::transform<std::complex<float> const*, std::complex<float> const*, std::complex<float>*, std::multiplies<std::complex<float>>>	libsynthesis.so.3877.90.108	9.603s	12.0%
casacore::FFTServer<float, std::complex<float>>::fft0	libcasa_scimath.so.3877.90.108	7.259s	9.1%
casacore::arrays_internal::Storage<std::complex<float>, std::allocator<std::complex<float>>>::construct	libatnf.so.3877.90.108	4.724s	5.9%
func@0x18ad0	libgomp.so.1	3.860s	4.8%
__memmove_ssse3_back	libc.so.6	2.794s	3.5%
[Others]	N/A*	51.710s	64.7%

Figure 11: Top hotspots of WAsp

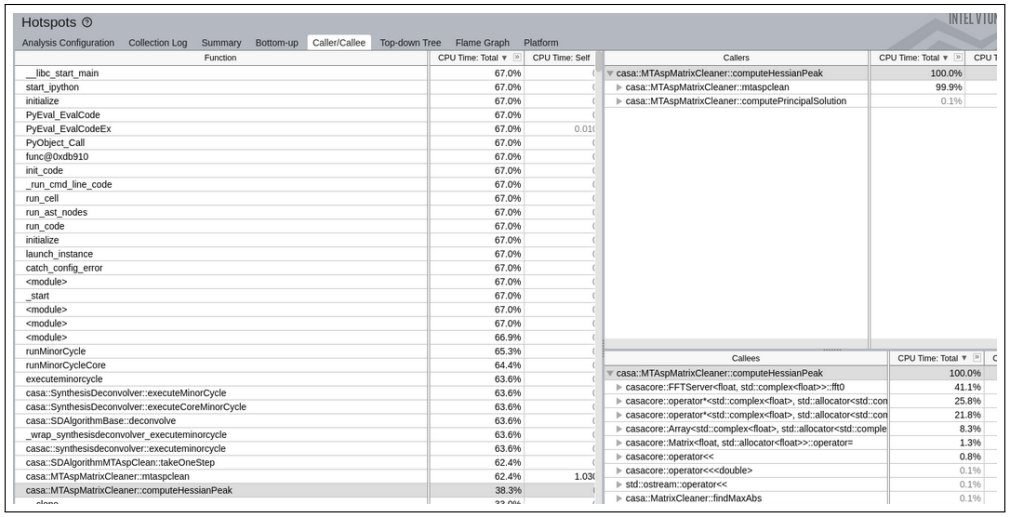


Figure 12: Top-down performance analysis of the WAsp hotspots

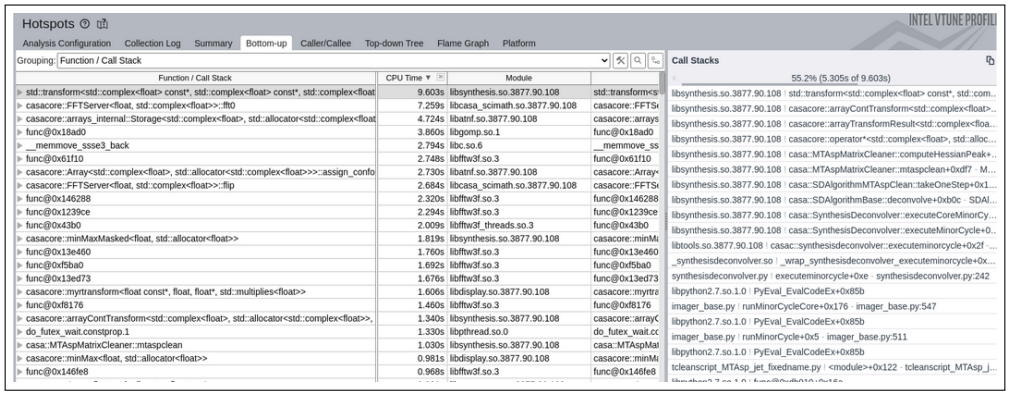


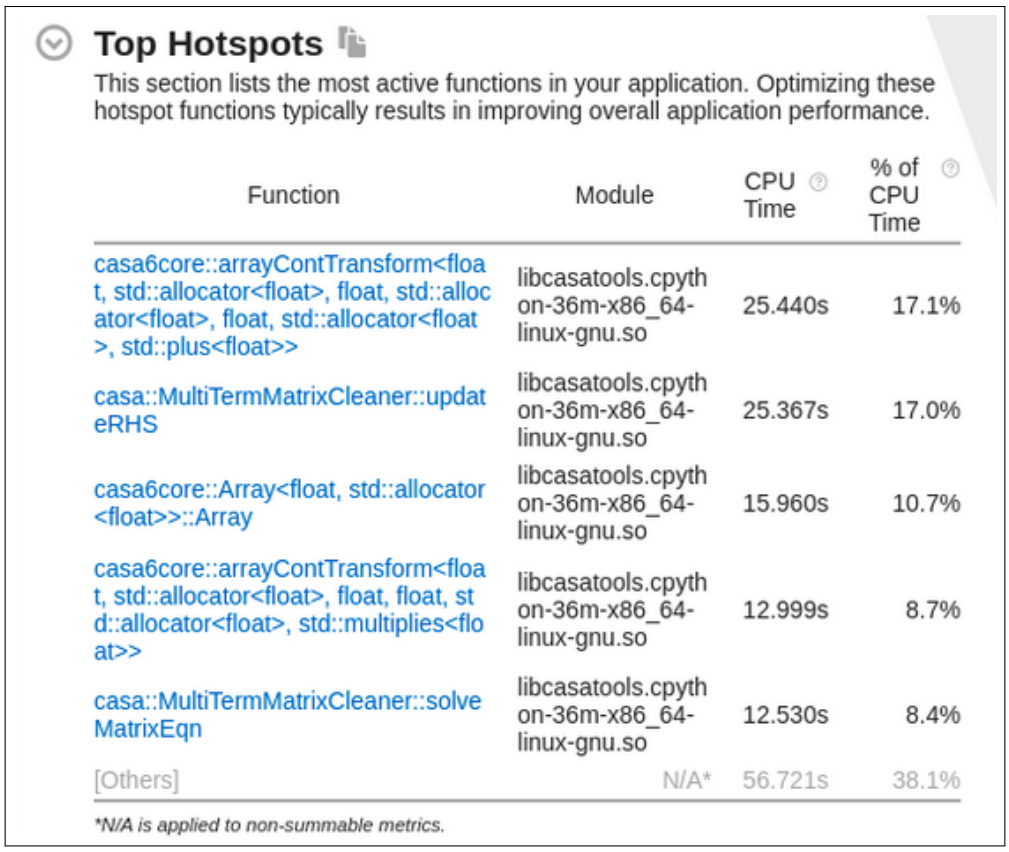
Figure 13: Bottom-up performance analysis of the `mtaspclean` function

2.2.4 MS-MFS

Hotspots analysis in the Figures 14 and 15 show that 92% of the MS-MFS runtime is spent on `mtclean`. 37% of `mtclean` is spent on `updateModelAndRHS`, 33% is on `solveMatrixEqn`, and 28% is on `chooseComponent`. The `computeHessianPeak` or `casacore::FFTServer::fft0` are not hotspots in MS-MFS like WAsp. This is probably because compared with MS-MFS, WAsp has

- an additional `findMaxAbs` function in `computeHessianPeak` because of the nature of the asp algorithm.
- additional matrix resize functions in `computeHessianPeak` for filling up the two matrices.
- `getActiveSetAspen` to dynamically optimize scales at every iteration which is mostly `casacore::FFTServer::fft0` and `flip`.

On the other hand, WAsp only spent 8% of the runtime on `updateModelAndRHS` and 3% on `solveMatrixEqn` because it only involves two scales (0-scale and the optimal scale), instead of five scales in MS-MFS.



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
casa6core::arrayContTransform<float, std::allocator<float>, float, std::allocator<float>, float, std::allocator<float>, std::plus<float>>	libcasatools.cpython-36m-x86_64-linux-gnu.so	25.440s	17.1%
casa::MultiTermMatrixCleaner::updateRHS	libcasatools.cpython-36m-x86_64-linux-gnu.so	25.367s	17.0%
casa6core::Array<float, std::allocator<float>>::Array	libcasatools.cpython-36m-x86_64-linux-gnu.so	15.960s	10.7%
casa6core::arrayContTransform<float, std::allocator<float>, float, float, std::allocator<float>, std::multiplies<float>>	libcasatools.cpython-36m-x86_64-linux-gnu.so	12.999s	8.7%
casa::MultiTermMatrixCleaner::solveMatrixEqn	libcasatools.cpython-36m-x86_64-linux-gnu.so	12.530s	8.4%
[Others]	N/A*	56.721s	38.1%

**N/A is applied to non-summable metrics.*

Figure 14: Top hotspots of MS-MFS

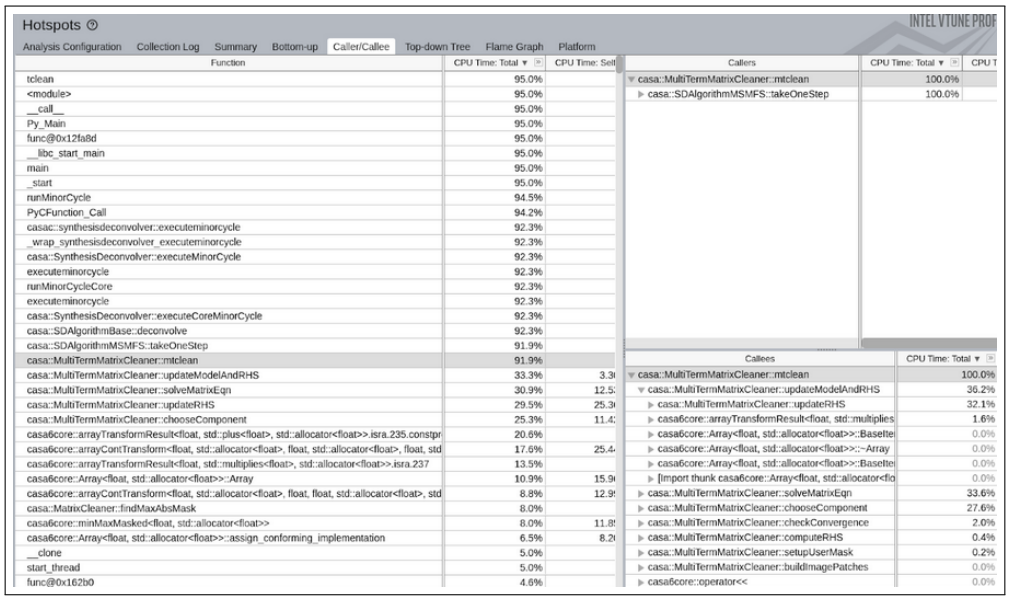
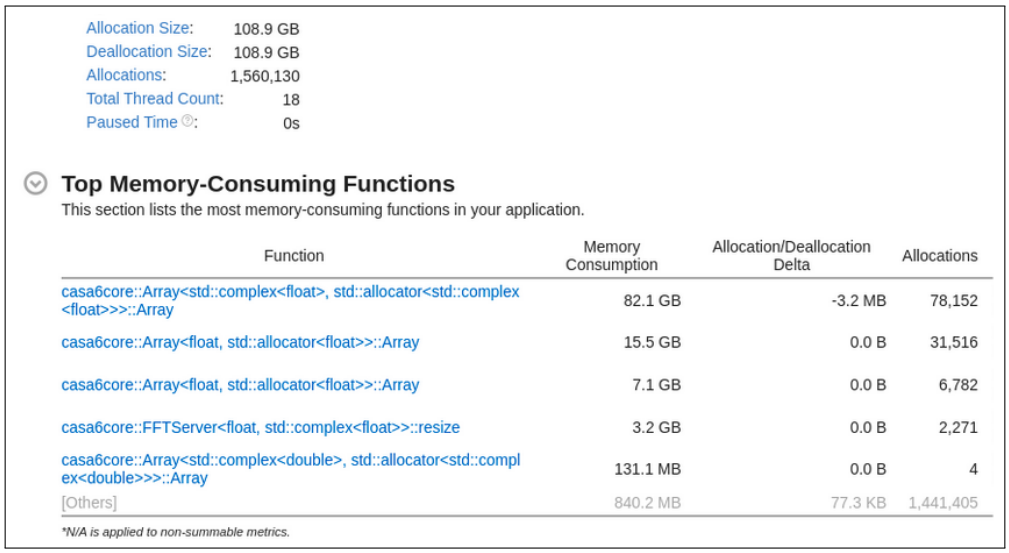


Figure 15: Top-down performance analysis of the MS-MFS hotspots

2.3 Memory Consumption

2.3.1 AspClean

The memory consumption analysis of AspClean in Figures 16 and 17 below shows that the memory consumption is mainly in the `getActiveSetAspen` function for matrix allocation, resize, and `casacore::FFTServer::fft0` and `flip`. The total memory consumption is about 109 GB and the peak memory consumption is about 180 MB.



Allocation Size: 108.9 GB
Deallocation Size: 108.9 GB
Allocations: 1,560,130
Total Thread Count: 18
Paused Time: 0s

Top Memory-Consuming Functions

This section lists the most memory-consuming functions in your application.

Function	Memory Consumption	Allocation/Deallocation Delta	Allocations
casa6core::Array<std::complex<float>, std::allocator<std::complex<float>>>::Array	82.1 GB	-3.2 MB	78,152
casa6core::Array<float, std::allocator<float>>::Array	15.5 GB	0.0 B	31,516
casa6core::Array<float, std::allocator<float>>::Array	7.1 GB	0.0 B	6,782
casa6core::FFTServer<float, std::complex<float>>::resize	3.2 GB	0.0 B	2,271
casa6core::Array<std::complex<double>, std::allocator<std::complex<double>>>::Array	131.1 MB	0.0 B	4
[Others]	840.2 MB	77.3 KB	1,441,405

*N/A is applied to non-summable metrics.

Figure 16: Total and top memory consumption of AspClean

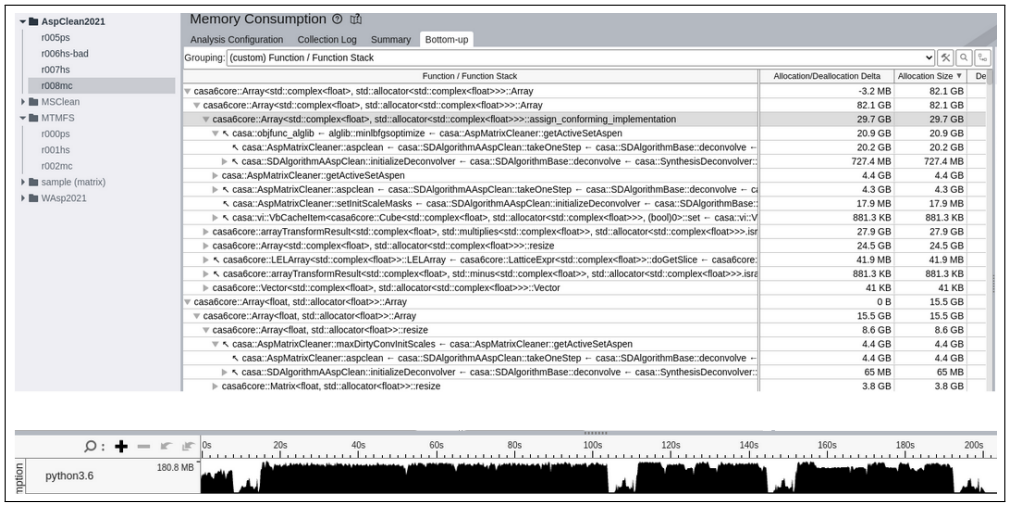


Figure 17: Bottom-up and peak memory consumption of AspClean

2.3.2 MS-Clean

Figure 18 show that the total memory consumption of MS-Clean is about 72 GB and the peak memory consumption is about 346MB. AspClean memory consumption is 1.5x more than MS-Clean (109GB vs. 72GB). This is probably because of the `getActiveSetAspen` function in AspClean which consists

of additional matrix allocation, resize, and `casacore::FFTServer::fft0` and `flip`. The memory consumption of MS-Clean is mostly on `MatrixCleaner::clean()`.

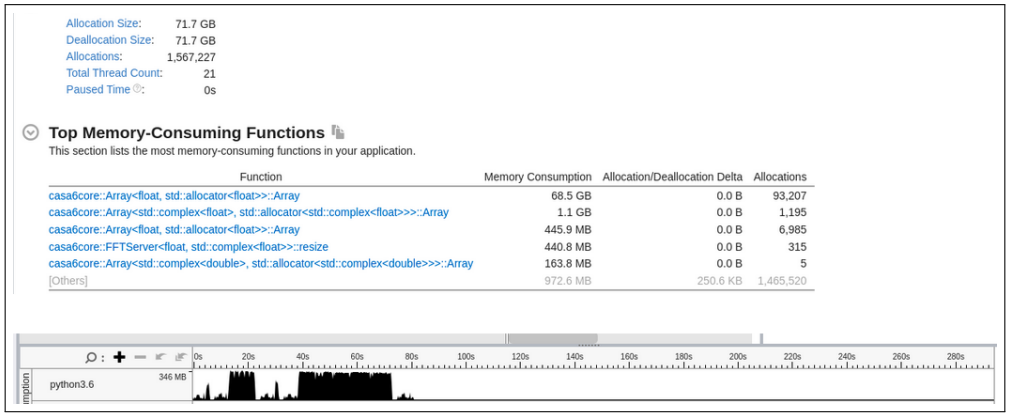


Figure 18: Memory consumption analysis of MS-Clean

2.3.3 WAsp

Figures 19 and 20 show that the total memory consumption of WAsp is about 53 GB and the peak memory consumption is about 127MB. WAsp memory usage is 52% of AspClean (53GB vs. 109GB). More investigations are required to understand why AspClean uses about 20GB for the optimization function, `objfunc_alglib`, but WAsp only uses about 1 GB since both call the identical `objfunc_alglib` function. Most of the memory consumption of WAsp (about 38GB) is on `computeHessianPeak`.

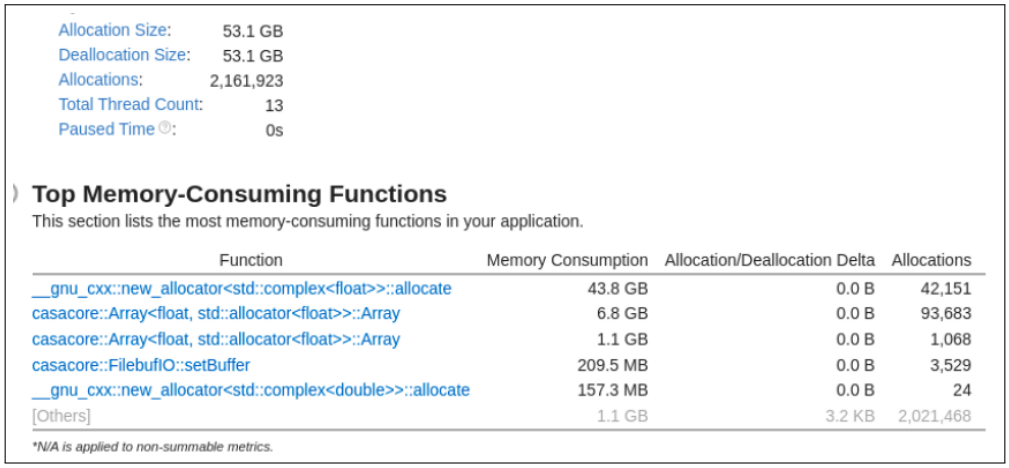


Figure 19: Total and top memory consumption of WAsp

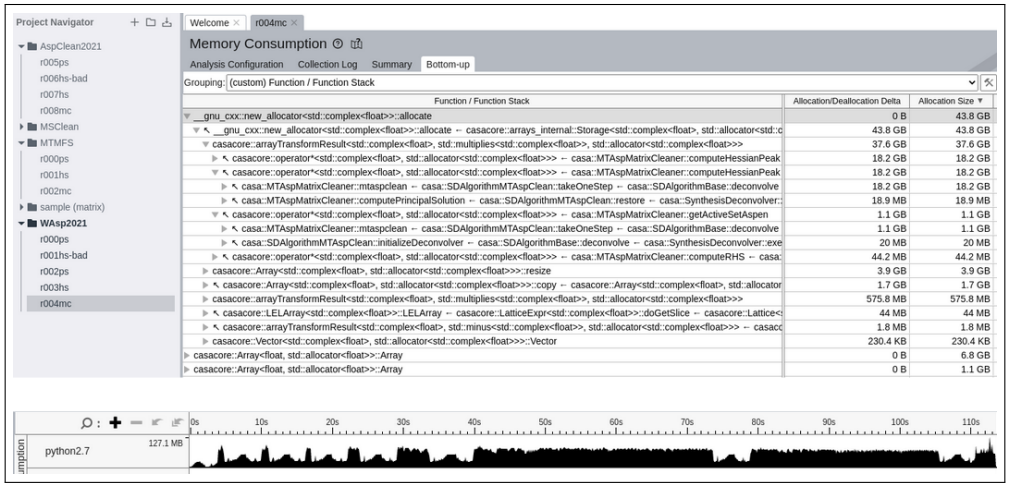


Figure 20: Bottom-up and peak memory consumption of WAsp

2.3.4 MS-MFS

Figures 21 and 22 show that the total memory consumption of MS-MFS is about 203 GB and the peak memory consumption is about 188 MB. The majority of the MS-MFS memory consumption is on `updateRHS`, `updateModelAndRHS`, and `solveMatrixEqn`, etc. WAsp memory usage is about 26% of MS-MFS's (53 GB vs. 203 GB) because it only needs to allocate memory for two scales instead of five scales in MS-MFS.

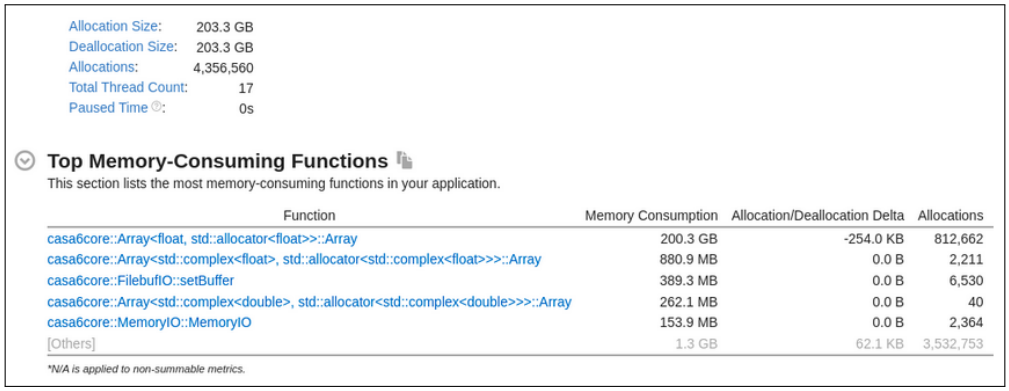


Figure 21: Total and top memory consumption of MS-MFS

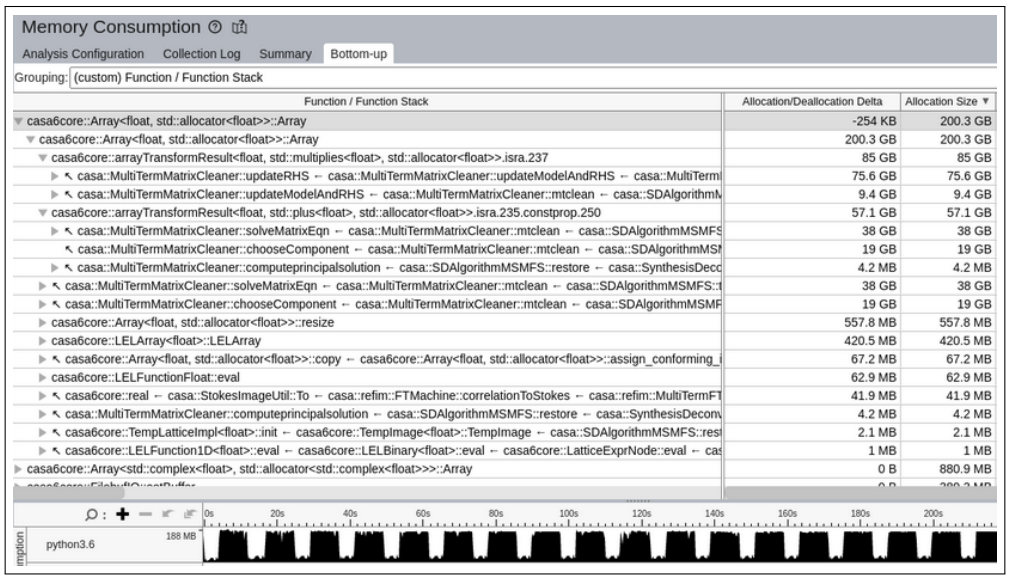


Figure 22: Bottom-up and peak memory consumption of MS-MFS

3 Analysis

For AspClean, the most expensive function is `getActiveSetAspen` that dynamically optimizes scales at every minor cycle. 83% of AspClean runtime is on `getActiveSetAspen`. 80% of the `getActiveSetAspen` runtime is on the BFGS optimization function. Almost all of optimization function runtime is on the *fdf* calculation (CASA code that consists of FFT (83%) and matrix operations (17%), etc). This suggests the performance bottleneck of AspClean is FFT, and a GPU-based FFT may improve the runtime performance. On the other hand, for the BFGS optimization, moving matrix calculations to GPU may not be necessary because of the following analysis of matrix calculations vs. FFT runtime.

BFGS Asp matrix construct runtime 897 us
BFGS *dAdS* matrix calculation runtime 281 us
BFGS FFT runtime 206928 us

which shows matrix calculation is considered cheap compared to FFT. Section 3.1 summarizes the key findings of this performance analysis.

3.1 Key Findings

- For AspClean
  - 70% of `aspclean` is on `casacore::FFTServer::fft0/flip`; 30% of `aspclean` is on matrix operations that are not localized in one function.
  - For BFGS optimization, matrix calculation (17%) is considered cheap compared to FFT (83%).
  - Therefore, the performance bottleneck is at FFT, which is not seen in MS-Clean. A GPU-based FFT may improve the runtime performance.
- For WAsp
  - 62% of `mtaspclean` is spent on `computeHessianPeak`, 41% of which is on `casacore::FFTServer::fft0` and 59% is on matrix operations.

- 17% of mtaspclean is spent on `getActiveSetAspen`, 73% of which is on `casacore::FFTServer::flip/fft0` and the rest 27% is on matrix operations.
- In summary, 38% of the WAsp runtime is spent on `casacore::FFTServer::fft0/flip` and 41% is on matrix operations that are more localized in `computeHessianPeak`.
- Improve `findMaxAbs` may reduce `computeHessianPeak` runtime.
- Therefore, GPU may be used to speed up the matrix operations.

Tables 1 and 2 below summarize the computational performance and memory consumption of the four deconvolution algorithms in this analysis. Table 1 shows that AspClean/WAsp may be about 2x slower than baseline reference, MS-Clean/MS-MFS. This is quite manageable even for ngVLA and we expect utilizing GPU can further improve the runtime performance. Section 4.2 shows that our preliminary implementation of using the GPU-based FFT (i.e. `cuFFT`) in AspClean results in at least 2x speedup. This demonstrates utilizing GPU for the hotspot functions of deconvolution algorithms is a worthwhile line of work to pursue for ngVLA.

Table 1: Computational Performance (jet dataset)

	AspClean	MS-Clean	WAsp	MS-MFS
number of iterations	2000	10000	4000	10000
gain	0.8	0.2	0.6	0.1
Metric (number of iterations x gain)	1600	2000	2400	1000
runtime	2 mins 29 sec to 4 mins	2 mins 12 sec	4 mins 48sec	3 mins 34sec

Table 2: Memory consumption

	AspClean	MS-Clean	WAsp	MS-MFS
Total memory usage (GB)	109	72	53	203
Peak memory consumption (MB)	180	346	127	188

Table 2 shows that the WAsp total memory usage is about 26% of MS-MFS (53 GB vs. 203 GB) and is about 52% of AspClean (53 GB vs. 109 GB). Peak memory consumption is also analyzed here because of the following reasons. The two attributes of memory system performance are generally bandwidth and latency. Many techniques have been used to improve the performance of the memory systems of computers for high performance computing. Some memory system design changes improve one at the expense of the other, and other improvements positively impact both bandwidth and latency. Bandwidth generally focuses on the best possible steady-state transfer rate of a memory system. Usually this is measured while running a long unit-stride loop reading or reading and writing memory. Latency is a measure of the worst-case performance of a memory system as it moves a small amount of data such as a 32- or 64-bit word between the processor and memory. The theoretical peak memory bandwidth can be calculated from the memory clock and the memory bus width. With a large enough cache, a small (or even moderately large) data set can fit completely inside and get incredibly good performance. One way to make the cache-line fill operation faster is to "widen" the memory system. For example, instead of having two rows of DRAMs, multiple rows of DRAMs can be used. Since both AspClean and WAsp have lower peak memory consumption than MS-Clean and MS-MFS, this is an improvement on the memory performance which suggests we may not need an advanced memory system for deconvolution for ngVLA.

4 Performance Improvement from cuFFT

4.1 Stand-alone program

Since FFT is the bottleneck of the AspClean performance efficiency, a stand-alone CUDA program was developed to compare the performance of CASA's `casacore::FFTServer::fft0` (based on `fftw`)

and CUDA's `cuFFT`. The program calculates the total runtime of `cuFFT` which includes moving an image from CPU to GPU, doing a forward FFT by `cuFFT`, and then moving the image back to CPU. Research shows that the CPU-based FFT approach (e.g. `fftw`) is highly optimized for input sizes that can be written in the form $2a \times 3b \times 5c \times 7d$. On the other hand, GPU-based FFT (i.e. `cuFFT`) shows an FFT runtime difference of up to one order of magnitude for large input signals of power of 2 and odd shape type.

Our study (Figure 23) compares the runtime of the `casacore::FFTServer::fft0` (red) and `cuFFT` (blue) for various image sizes. Kumar Golap earlier did an analysis on the runtime of FFTW 3.0 via `FFT2D` wrapper, and observed that there is a distinct bump in slowness at image size of 4096 as compared to 4000 and 5000. It is included (brown) and shown in Figure 23 as well. The `FFT2D` wrapper was implemented in CASA because as the images become big, it is not the `fftw` part in `FFTServer::fft0` that dominates, but the `memcpy` which is serial in that implementation. Therefore, the `FFT2D` wrapper was implemented with multithreaded `fftshift` and `memcopies`.

The result in Figure 23 shows that, with I/O time taken into consideration, `cuFFT` is faster than `fftw` when image size is > 2048. `cuFFT` is also faster than `FFT2D`. The `cuFFT` itself, not considering data movement, is much faster (about 8x) even for small images, so when more advanced high performance computing architecture is available, we can see more performance improvements from using `cuFFT`.

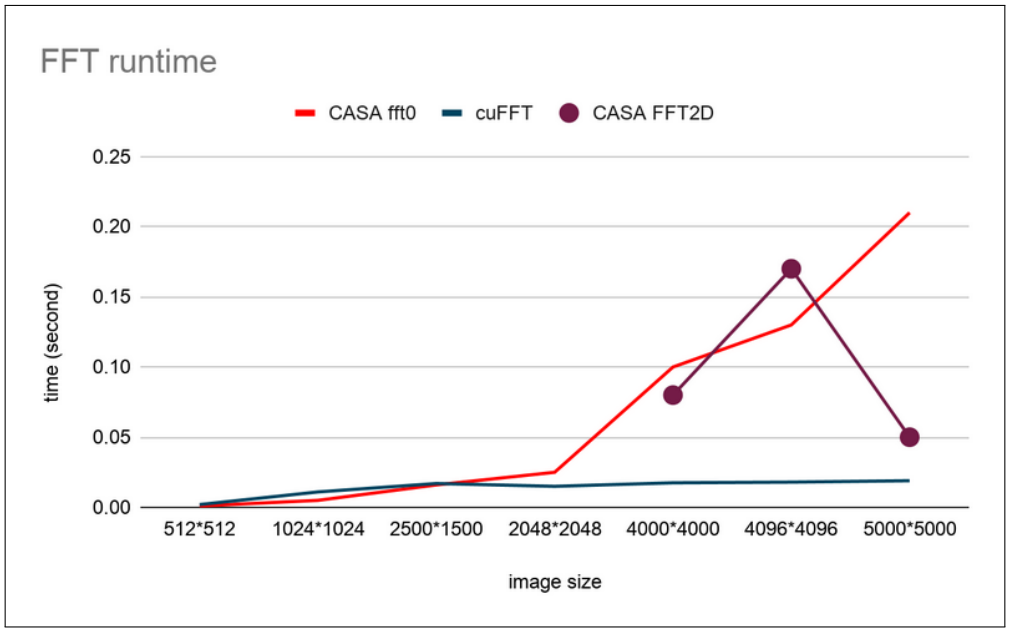


Figure 23: Runtime of `cuFFT` vs. `casacore::FFTServer::fft0`

4.2 `cuFFT` in AspClean

The stand-alone program described in Section 4.1 already shows `cuFFT` is faster than `fftw` for large image sizes with considerations of data transfer between CPU and GPU. We next evaluated whether `cuFFT` can also improve the AspClean computational performance in CASA. We utilized the CASA build system for the HPG/roadrunner project that has CUDA dependencies and made additional changes to make CASA able to use `cuFFT`. As a proof-of-concept study, we started with only replacing the `casacore::FFTServer::fft0` in the most time consuming BFGS optimization function in AspClean with `cuFFT`. This preliminary implementation shows a 40%-45% runtime reduction in the BFGS optimization, which is equivalent to about 30% runtime reduction of the total AspClean runtime. This 2x speedup in the hotspot function of AspClean is only a lower-limit. More performance improvement is expected when every `fft0` in AspClean is replaced by `cuFFT`. It is worth noting that this prelimi-

nary study is conducted on the jet dataset which has an image size of 512x512. Based on the result of the stand-alone program, `fft0` should be faster than `cuFFT` for this small image size. However, the preliminary implementation demonstrates `cuFFT` in AspClean can improve runtime by at least 2x even for small image sizes with the data transfer between memories factored in.

5 Conclusions

The primary goal of the work described in this memo is to identify issues in application of (W)Asp deconvolution algorithms to ngVLA. The necessary prerequisite for this are:

- Determine the computing hot-spots in (W)Asp implementation and explore ways to deploy those on a GPU for speed up for ngVLA.
- Analyze the memory consumption of the (W)Asp implementation and identify issues when running on the ngVLA computing architecture.
- As a baseline reference, give a short comparison with MS-Clean and MS-MFS of cost of computing and imaging performance.

In this work, we did a detailed performance analysis of all four deconvolution algorithms and identified the most-time consuming functions of the AspClean and WAsp. Based on this analysis, we suggested a couple of ways to utilize GPU for speedup for ngVLA. As a proof-of-concept study, we completed a preliminary implementation of using `cuFFT` in AspClean. This preliminary implementation gives at least 2x runtime speedup and demonstrates that utilizing GPU for the hotspot functions of deconvolution algorithms is a worthwhile line of work to pursue with data transfer between CPU-GPU memories factored in.

In summary, this work provides the following insights for applying (W)Asp to ngVLA.

- AspClean and WAsp provides better imaging performance and are about 2x slower than MS-Clean/MS-MFS in the current implementation running in serial on CPU.
- The performance bottleneck of AspClean is at FFT, while matrix operations in AspClean are considered cheap. Thus, a GPU-based FFT may improve the AspClean runtime performance for ngVLA.
- As a proof-of-concept study, an initial implementation that replaces the `casacore::FFTServer::fft0` in the most time consuming BFGS optimization function in AspClean with `cuFFT` already results in 40%-45% runtime reduction. It is worth noting that this is only a lower-limit on the speedup. When every `fft0` in the AspClean is replaced with `cuFFT`, more performance improvement is expected.
- For WAsp, 38% of the runtime is spent on `casacore::FFTServer::fft0/flip` and 41% is on matrix operations that are more localized in `computeHessianPeak`. Therefore, the matrix operations in `computeHessianPeak` may be moved to GPU for speedup for ngVLA.
- Both AspClean and WAsp have lower peak memory consumption than MS-Clean and MS-MFS. This is an improvement on the memory performance which suggests we may not need an advanced memory system for applying (W)Asp to ngVLA.

References

[1] Intel vtune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.vk69qb.

[2] S. Bhatnaga, R. Hiriart, and M. Pokorny. Size-of-computing estimates for ngvla synthesis imaging. Technical report, ngVLA Memo 4, Aug. 2021. URL: "https://library.nrao.edu/public/memos/ngvla/NGVLAC_04.pdf".

[3] M. Hsieh and S. Bhatnaga. Efficient adaptive-scale clean deconvolution in casa for radio interferometric images. Technical report, Jan. 2021. URL: "https://safe.nrao.edu/wiki/pub/Software/Algorithms/WebHome/ardg_aspclean.pdf".