Efficient Faraday Synthesis

W. D. Cotton (NRAO), January 31, 2025

***Abstract*—Computational efficiency of the software used for Faraday synthesis and analysis in the Obit system is discussed. Improved memory access patterns lead to dramatic, factor of 7 to 10, improvement in the runtime for Faraday syntheses and modest improvements in runtime for Faraday analysis. In all cases, the CPU requirements are significantly reduced. Tests used multi–threading and hand coded AVX intrinsics to boost performance. AVX512 seems to give little improvement over AVX on a machine with a 512 bit bus.**

***Index Terms*—Computational efficiency**

I. Introduction

FARADAY analysis of polarimetric images using a direct parameter search can be computationally expensive. The Faraday depth spectrum is essentially the Fourier transform of the Q+iU spectrum in λ^2 space. Since these spectra are never sampled regularly in λ^2 space, a direct rather than FFT transform is needed. This memo describes techniques for efficient Faraday synthesis and analysis in the Obit package [1][1].

The image level Faraday synthesis/analysis described in this memo is a good test case for exploring efficiency improvements as it is relatively simple and has no dependencies to complicate the logic. I.e. the operations on each pixel on the sky are independent of all others. Also, much of the work is done after the input data is read and before the results are written so that timing of the core functionality is not affected by the speed of the I/O. Obit uses multi–threading [2] and vector intrinsics (SSE, AVX) [3], [4] to boost performance.

II. Faraday Rotation

The phenomenon of the rotation of the angle of a polarized signal passing through a magnetized plasma, Faraday rotation, has long been understood. The amount of this rotation [5] is:

$$\Delta\chi \;=\; \lambda^2\ 0.81 \int n_e B_{\|} dr, \tag{1}$$

where λ is the wavelength in m, n_e is the electron density in cm^{-3}, $B_{\|}$is the strength of the component of the magnetic field along the line of sight in μGauss and r is distance in parsec. This effect was further used to develop the concepts of Faraday dispersion and Faraday depth by [6] who introduced the technique of Faraday synthesis, derivation of the Faraday depth spectrum in a given sight-line. This technique has been further refined by [5], [7], [8], [9].

National Radio Astronomy Observatory, 520 Edgemont Rd., Charlottesville, VA, 22903 USA email: bcotton@nrao.edu

[1]http://www.cv.nrao.edu/∼bcotton/Obit.html

The Faraday spectrum is approximated using the Fourier series [10]

$$F_k(x,y) \;=\; K \sum_{j=1}^{n} W_j\ e^{-2i\phi_k(\lambda_j^2-\lambda_0^2)}[Q_j(x,y)+iU_j(x,y)] \tag{2}$$

for Faraday depth ϕ_k where W_j is the weight for frequency sub-band j of n, λ_j is the wavelength of frequency sub-band j, λ_0 is the reference wavelength, i is $\sqrt{-1}$ and Q_j and U_j are the Stokes Q and U sub-band images at frequency j. The normalization factor K is $1/\sum_{j=1}^{n} W_j$. W_j may also include a correction for spectral index[2], α:

$$W_j \;=\; w_j e^{-\alpha \log(\nu_j/\nu_0)} \tag{3}$$

where ν_j is the frequency of channel j, ν_0 is the reference frequency and the weight for sub-band j, w_j, is zero for frequency bins (nearly) totally blanked due to RFI filtering, one otherwise.

In Obit, the derivation of a Faraday cube (RA, Dec, Faraday depth) is done in task RMSyn. This is described in more detail in [10], [9] which also explore some of the limitations of the technique.

III. Parallel Computing in Obit

Multithreading is implemented using gthread pools[2]. Use of thread pools allows, in principle, reducing the overhead of starting and stopping threads which can be substantial. All threads in a thread pool are constrained to use the same function. Since a thread is not terminated when a function call finishes, this completion is indicated by an event which is handled by the thread pool manager. This mechanism prohibits multiple simultaneous thread pools. or multiple functions in the same thread pool.

Modern processors all support vector operations with vectors of length the width of the memory bus, currently 256 or 512 bits. This allows operations on vectors of 8 or 16 floats at the same cost of a single, scalar, operation. Compilers are (slowly) getting to use some of this functionality but for the ultimate boost in performance, use of "intrinsics" (https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html) is better. Hand coded libraries of common functions (e.g.sincos) are available to give a boost to these functions.

Intrinsics functions are essentially assembly level instructions in the form of c function calls. Obit has explicit support for SSE (128 bit = 4 float), AVX (256 bit = 8 float), AVX2 (256 bit with a stronger instruction set) and AVX512f (512 bit = 16 floats) vector operations [3], [4]. The supported vector

[2]The spectral index is defined as $I_\nu \propto \nu^\alpha$.

sets are chosen at compile time using options -DHAVE_SSE, -DHAVE_AVX, -DHAVE_AVX2, -DHAVE_AVX512 to select Obit code wrapped in #ifdefs and the options -msse, -mavx, -mavx2 and -m512f to allow the compiler to use the associated vector instruction sets.

IV. Previous Obit Implementation

As described in Section III, Obit makes extensive use of parallelization using both multi–threaded operation and vectorization to improve performance. Many operations on image–like data are implemented in the ObitFArray (float) and ObitCArray (complex) classes which provide a variety of operations on arrays of floats or complex values (as float pairs). Many of these function use a combination of multi-threading and SSE/AVX vectorization. For threaded functions, thread load balancing is done by dividing each array operation into equal number of pixels for each thread. This is possible as these operations do not include dependencies between pixels.

Since most of the array functions in classes ObitFArray and ObitCArray are fairly generic, they support magic value pixel blanking; a special value, similar to a NaN, is used to indicate no value [3]. This adds an additional computing expense but is simple to implement using intrinsics functions. The Fourier synthesis given in Equation 2 is implemented using image plane functions in the ObitFArray and ObitCArray classes. Some of these functions have multithreaded implementations which do their own thread management meaning the thread pools are repeatedly started and shut down.

A. Original RMSyn Faraday Synthesis

Task RMSyn reads arrays of ObitFArrays containing the observed Stokes Q and U cubes and generates a cube of complex image planes at a set of Faraday depths using Equation 2. A complex CLEAN in each pixel is optional but the focus of the current memo is the generation of the "dirty" cube. For each Faraday depth (ϕ) and λ^2:

1) **Convert Q and U FArrays to a CArray.** This operation uses ObitCArrayComplex.
2) **Rotation array.** Using ObitCArrayFill (threaded) fill a CArray with the complex value

$$e^{-2i\phi(\lambda^2-\lambda_0^2)}$$

   where λ_0^2 is the reference λ^2 (near 0) and i is $\sqrt{-1}$.
3) **Multiply Q/U Array by Rotation array.** using ObitCArrayMul (AVX implementation and threaded).
4) **Accumulate.** Add the result of the previous step to the accumulation CArray using ObitCArrayAdd (AVX implementation and threaded).

While this implementation is reasonably optimized it still involves extra copies of the Q/U data (converting to a CArray), uses a complex array (Rotation array) where a complex scalar is adequate and multiple passes are made through the data. Threads are only used for a single function call.

[3]This usage in AIPS predates the development of IEEE standards for floating point data.

V. Enhanced Obit RMSyn Implementation

Many of the inefficiencies in the original implementation of RMSyn are reduced using new ObitCArray function ObitCArraySMulAccum and implementating a more efficient way of managing threads. This enhanced implementation is embedded in the class ObitFaraSyn. These improvements are discussed in the following sections.

A. ObitCArray function ObitCArraySMulAccum

The operations described in Section IV-A are combined into the new ObitCArray function ObitCArraySMulAccum. The performance is enhanced using both AVX vectorization and multi–threading. This routine is given a Q and U plane from the input cubes and the complex scalar to rotate them for a given Faraday depth. The complex product is accumulated into the Faraday depth plane. The function invoked per thread is given in Figures V-A and V-A (really 1 & 2). Each thread is given a range of elements in a 1D representation of the 2D planes in the Q and U cubes.

The main complication is that neither c nor the AVX intrinsics support complex data types which must be implemented as operations on floats. AVX functions are clumsy at reorganizing data inside vectors so gather/scatter was used for the AVX512 update of the accumulation array and an even cruder implementation was used for AVX. AVX512 works on blocks of 16 floats and AVX on blocks of 8; any left over elements must be handled from straight c. The threading implementation uses a minimum or 50,000 elements per thread.

Since ObitCArraySMulAccum is an implementation for a particular application for which the replacement of pixel blanking with zeroes is possible, the routine does not need to check for pixel blanking. This is not the case for the generic array functions which must check for blanked pixels.

B. Thread Pool Management

Threading in Obit uses gthread thread pools to reduce the overhead of starting multiple threads. Each thread can be sent a sequence of jobs and the threads, once activated, remain alive until explicitly terminated. Unfortunately, the functions in the ObitFArray and ObitCArray classes do thread management independently for each call and divides the work up into a number of threads which are each executed once. When the same function is repeatedly called, a more efficient implementation would be to have the external software do the thread management and only start and stop the pool of threads once. To make best use of this, a function such as ObitCArraySMulAccum which combines multiple threaded operations, is needed.

Previously, thread management was only done internally in classes with threaded operations, e.g. ObitFArray and ObitCArray. Function ObitCArraySMulAccum (see Section V-A) is implemented in class ObitCArray which has been modified to allow external thread management. Several functions are now defined:

- **ObitCArrayMakeCAFuncArgs**

  /**

```
/**
 * Form complex array from two FArrays, multiply by complex scalar and
 * complex accumulate. Magic value blanking not supported.
 * Callable as thread
 * \param arg Pointer to CAFuncArg argument with elements:
 * \li FA_1     Real part of input
 * \li FA_2     Imaginary part of input
 * \li arg1     Real/Imaginary parts of scalar
 * \li out      Output ObitCArray accumulator
 * \li first    First element (1-rel) number
 * \li last     Highest element (1-rel) number
 * \li ithread  thread number, <0 -> no threading
 * \return NULL
 */
static gpointer ThreadCASMulAccum (gpointer arg)
{
  /* Get arguments from structure */
  CAFuncArg *largs = (CAFuncArg*)arg;
  ObitFArray *Fin_r     = largs->FA_1;
  ObitFArray *Fin_i     = largs->FA_2;
  ofloat     *cscalar   = largs->arg1;
  ObitCArray *Accum     = largs->out;
  olong      loElem     = largs->first-1;
  olong      hiElem     = largs->last;
  ofloat tr1, ti1, tr2, ti2;
  olong i, ilast;
  ofloat *rArr = Fin_r->array, *iArr = Fin_i->array, *oArr = Accum->array;
#if HAVE_AVX512==1
  CV16SF v1r, v1i, sr, si, tv1, tv2, tv3, tv4;
  V16SI vindex;
#elif HAVE_AVX==1
  olong j;
  CV8SF  v1r, v1i, sr, si, tv1, tv2, tv3, tv4;
#endif

  if (hiElem<loElem) goto finish;
  tr2 = cscalar[0]; ti2 = cscalar[1];
#if HAVE_AVX512==1  /* AVX 512 Vector (16 float) */
  sr.v = _mm512_set1_ps(tr2); /* Scalar real */
  si.v = _mm512_set1_ps(ti2); /* Scalar imaginary */
  vindex = _mm512_set_epi32(30,28,26,24,22,20,18,16,14,12,10,8 6,4,2,0);
  for (i=loElem; i<hiElem-16; i+=16) {
    v1r.v = _mm512_loadu_ps(&rArr[i]); /* Input reals */
    v1i.v = _mm512_loadu_ps(&iArr[i]); /* Input imaginaries */
    tv1.v = _mm512_mul_ps (v1r.v, sr.v);  /* tr1*tr2 */
    tv2.v = _mm512_mul_ps (v1i.v, si.v);  /* ti1*ti2 */
    tv1.v = _mm512_sub_ps (tv1.v, tv2.v); /* now Real part */
    tv3.v = _mm512_i32gather_ps(vindex,(void const*)(&oArr[i*2]),4); /* Gather reals */
    tv3.v = _mm512_add_ps (tv3.v, tv1.v);  /* Update reals */
    _mm512_i32scatter_ps((void*)(&oArr[i*2]),vindex, tv3.v,4); /* Scatter reals */
    tv3.v = _mm512_mul_ps (v1i.v, sr.v);  /* ti1*tr2 */
    tv4.v = _mm512_mul_ps (v1r.v, si.v);  /* tr1*ti2 */
    tv2.v = _mm512_add_ps (tv3.v, tv4.v); /* now Imaginary part */
    tv3.v = _mm512_i32gather_ps(vindex,(void const*)(&oArr[1+(i*2)]),4); /* Gather imag */
    tv3.v = _mm512_add_ps (tv3.v, tv2.v);  /* Update imag */
    _mm512_i32scatter_ps((void*)(&oArr[1+(i*2)]),vindex, tv3.v,4); /* Scatter imag */
  } /* end outer loop */
  ilast = i;  /* How far did I get? */
```

Fig. 1. ThreadCASMulAccum function

```
#elif HAVE_AVX==1  /* AVX Vector (8 float) */
  sr.v = _mm256_broadcast_ss(&tr2); /* Scalar real */
  si.v = _mm256_broadcast_ss(&ti2); /* Scalar imaginary */
  for (i=loElem; i<hiElem-8; i+=8) {
    v1r.v = _mm256_loadu_ps(&rArr[i]); /* Input reals */
    v1i.v = _mm256_loadu_ps(&iArr[i]); /* Input imaginaries */
    tv1.v = _mm256_mul_ps (v1r.v, sr.v);  /* tr1*tr2 */
    tv2.v = _mm256_mul_ps (v1i.v, si.v);  /* ti1*ti2 */
    tv1.v = _mm256_sub_ps (tv1.v, tv2.v); /* now Real part */
    /* Gather/scatter the hard way */
    tv3.v = _mm256_set_ps(oArr[(i+7)*2],oArr[(i+6)*2],oArr[(i+5)*2],oArr[(i+4)*2],
  oArr[(i+3)*2],oArr[(i+2)*2],oArr[(i+1)*2],oArr[i*2]);
    tv3.v = _mm256_add_ps (tv3.v, tv1.v);  /* Update reals */
    for (j=0; j<8; j++) oArr[(i+j)*2] = tv3.f[j]; /* Yuck */
    tv3.v = _mm256_mul_ps (v1i.v, sr.v);  /* ti1*tr2 */
    tv4.v = _mm256_mul_ps (v1r.v, si.v);  /* tr1*ti2 */
    tv2.v = _mm256_add_ps (tv3.v, tv4.v); /* now Imaginary part */
    tv3.v = _mm256_set_ps(oArr[1+(i+7)*2],oArr[1+(i+6)*2],oArr[1+(i+5)*2],oArr[1+(i+4)*2],
  oArr[1+(i+3)*2],oArr[1+(i+2)*2],oArr[1+(i+1)*2],oArr[1+i*2]);
    tv3.v = _mm256_add_ps (tv3.v, tv2.v);  /* Update imag */
    for (j=0; j<8; j++) oArr[1+(i+j)*2] = tv3.f[j]; /* Yuck */
  } /* end outer loop */
  ilast = i;  /* How far did I get? */
#else /* Scalar  */
  ilast = 0;  /* Do all */
#endif
  /* Loop over whatever is left over */
  for (i=ilast; i<hiElem; i++) {
    tr1 = rArr[i]; ti1 = iArr[i];
    oArr[i*2]   += tr1*tr2 - ti1*ti2;
    oArr[1+i*2] += ti1*tr2 + tr1*ti2;
  } /* end loop over array */

  /* Indicate completion */
  finish:
  if (largs->ithread>=0)
    ObitThreadPoolDone (largs->thread, (gpointer)&largs->ithread);

  return NULL;
} /* end ThreadCASMulAccum */
```

Fig. 2. ThreadCASMulAccum function cont'd

```
    * Make arguments for a Threaded function
    * For use outside the ObitCArray class
    * \param thread     ObitThread object
    * \param in         CA to be operated on
    * \param in2        2nd CA
    * \param out        output CA
    * \param FA_1       First FArray
    * \param FA_2       Second FArray
    * \param FA_3       Third FArray
    * \param FA_4       Fourth FArray
    * \param larg1      Length of arg1 (float)
    * \param larg2      Length of arg2 (float)
    * \param larg3      Length of arg3 (float)
    * \param larg4      Length of arg4 (float)
    * \param larg5      Length of arg5 (float)
    * \param larg6      Length of arg6 (float)
    * \param larg7      Length of arg7 (float)
    * \param ThreadArgs [out] CAFuncArg array
    *   delete with ObitCArrayKillCAFuncArgs
    * \return number of elements in args
        (number of allowed threads).
    */
olong ObitCArrayMakeCAFuncArgs
   (ObitThread *thread, ObitCArray *in,
    ObitCArray *in2, ObitCArray *out,
    ObitFArray *FA_1, ObitFArray *FA_2,
    ObitFArray *FA_3, ObitFArray *FA_4,
    olong larg1, olong larg2,
```

```
    olong larg3, olong larg4,
    olong larg5, olong larg6,
    olong larg7, gpointer *ThreadArgs)
```

This takes a number of application specific parameters, determines the number of threads allowed and creates an array of opaque thread argument arrays (ThreadArgs) and returns the maximum number of allowed threads. The number of threads and the ThreadArgs array pointer can be passed to the threaded application.

- **ObitCArrayKillCAFuncArgs**

```
/**
 * Delete arguments for ThreadCAFunc
 * \param nThreads   number of threads
 * \param ThreadArgs array of CAFuncArg
 */
void ObitCArrayKillCAFuncArgs
  (olong nThreads, gpointer ThreadArgs)
```

This function stops the threads in the thread pool and frees components in ThreadArgs.

- **ObitCArraySMulAccum**

```
/**
 *  Form complex array from two FArrays,
 *  multiply by complex scalar and
 *  complex accumulate.
 * \param Fin_r   Input real FArray
 * \param Fin_i   Input imaginary FArray
 * \param cscalar Complex Scalar [r,i]
 * \param Accum   CArray Accumulator
 */
void ObitCArraySMulAccum
  (ObitFArray* Fin_r,ObitFArray* Fin_i,
   ofloat cscalar[2],ObitCArray* Accum)
```

Version of the routine which manages threads internally.

- **ObitCArraySMulAccumTh**

```
/**
 *  Form complex array from two FArrays,
 *  multiply by complex scalar and
 *  complex accumulate.
 *  Threading controlled externally.
 * \param Fin_r   Input real FArray
 * \param Fin_i   Input imaginary FArray
 * \param cscalar Complex Scalar [r,i]
 * \param Accum   CArray Accumulator
 * \param nThreads   Number of threads
 * \param threadArgs Thread arguments
 */
void ObitCArraySMulAccumTh
  (ObitFArray* Fin_r,ObitFArray* Fin_i,
   ofloat cscalar[2],ObitCArray* Accum,
   olong nThreads,gpointer ThreadArgs)
```

Version of the routine for managing threads externally.

C. RMSyn Timing Comparison

Timing tests of the original RMSyn and the enhanced version were performed on machines Gandalf (sixteen Intel Xeon cores running @ 3.1 GHz and which supports SSE and AVX), Smeagle (twenty–four Intel Xeon Gold cores running @ 3.0 GHz and which supports SSE, AVX, AVX2, and AVX512 instructions) and Cheeta (seventy–two Intel Xeon cores running @ 2.1 GHz and which supports SSE, AVX, and AVX2). Each of these systems has 256 GByte of RAM. Gandalf and smeagle have software RAID-5 disks and cheeta has a 20 TByte SSD disk. Testing included a number of compilers including gcc 4.8.5, gcc 8.3.1 and icc 19.1.1.217; all compilations use -O3 optimization. All of the major computation code used was compiled with the same options for each test.

Two data sets were used, the smaller one with $2001\times2001\times200$ pixels ("Small") and a larger one $4882\times4882\times225$ pixels ("Large"). In both cases a substantial number of the Q/U channels were blanked due to RFI. For both tests, a Faraday depth cube with 1000 planes was formed. The elapsed wall clock and CPU times for just the Faraday synthesis part of runs of the old and new versions of RMSyn are given in Table I with the ratio of old to new wall clock and CPU times. The test data used are from MeerKAT at L band and are described in [11], [10], [9].

The more efficient newer implementation makes a dramatic difference in the wall clock (real) times, a factor of 7 for the older machines (galdalf, cheeta) and 10 for the newer (smeagle) one with more cores than gandalf and longer vectors.

In the new implementation, complex fill, multiply and sum is done in one pass through the data with no blank checking. Gather/scatter is used for AVX512 but is crudely emulated in AVX using SSE (4 float) operations for the "scatter" as they are more flexible. AVX2 has gather/scatter but that wasn't implemented or tested. The routines used in the old tests were generic and supported pixel blanking whereas in new tests, the routines were specific to that problem and any blanks were replaced by zeroes once and did not need testing. The most significant differences in the new and old implementations are that that the new version needs fewer passes through the data and doesn't need to check for blanked pixels.

Different compilers listed in Table I have different properties. One concern is the behavior of the tests, especially "Old", on smeagle with AVX512/AVX/no AVX ("no AVX" means the work was done in straight c but was allowed to use SSE instructions). The AVX512 and no AVX runs were comparable but AVX with gcc 8.3.1 took MUCH longer for both the old and new tests. The bulk of the time in the "Old" test should have been in ObitCArray:ThreadCAMul but the AVX512 and AVX implementations use the equivalent _mm512 and _mm256 functions. The same is also true of ObitCArray:ThreadCAAdd. The new implementations are also very similar for AVX512 and AVX. When the same AVX tests are rerun on smeagle compiling with either the older gcc 4.8.5 or icc, the results are comparable to the AVX512 runs. gcc 8.3.1 seems to produce particularly poorer results when using AVX than the older gcc 4.8.5; even when a large fraction of the work uses intrinsics. Using AVX512 shows little to no advantage over AVX; nor does icc over gcc.

Other than the discrepant results using gcc 8.3.1 on smeagle, the old tests show little variation in performance with vector

TABLE I
RMSYNTH CPU TIMING

System	Test	Vector	Compiler	Old Real min.	Old CPU min.	New Real min.	New CPU min.	Real Ratio	CPU Ratio
Gandalf	Small	AVX	gcc4.8.5	34.8	157.8	4.47	43.1	7.8	3.7
Gandalf	Large	AVX	”	219.0	943.0	27.8	290.4	7.8	3.2
Cheeta	Small	AVX	gcc4.8.5	39.5	314.7	3.25	117.9	12.2	2.7
Cheeta	Large	AVX	”	309.3	2775.6	35.8	1894.4	8.6	1.4
Cheeta	Small	no AVX	gcc4.8.5	38.2	311.7	49.0	1872.8	8.7	1.7
Cheeta	Large	no AVX	”	321.2	3408.0	414.0	15951.	0.8	0.2
Smeagle	Small	AVX512	gcc8.3.1	22.2	116.0	2.43	35.7	9.1	3.2
Smeagle	Large	AVX512	”	213.1	1255.8	20.4	402.3	10.4	3.1
Smeagle	Small	AVX512	icc	23.4	116.0	2.45	35.9	9.6	3.2
Smeagle	Large	AVX512	”	206.0	1177.2	20.6	401.7	10.0	2.9
Smeagle	Small	AVX	gcc8.3.1	44.5	211.2	4.57	71.0	9.7	3.0
Smeagle	Large	AVX	”	381.3	2110.7	29.1	465.1	13.1	4.5
Smeagle	Small	AVX	gcc4.8.5	22.0	89.3	2.12	34.2	10.4	2.6
Smeagle	Large	AVX	”	203.9	1166.7	20.0	400.9	10.2	2.9
Smeagle	Small	AVX	icc	21.3	109.0	2.15	34.1	9.9	3.2
Smeagle	Large	AVX	”	206.0	1243.5	20.1	400.2	10.2	3.1
Smeagle	Small	no AVX	gcc8.3.1	24.2	136.1	14.2	174.0	1.7	0.8
Smeagle	Large	no AVX	”	217.0	1399.9	99.6	1195.5	2.18	1.2

length. This suggests that the limiting factor was the larger number of passes through the data cubes in the old algorithm; much of the time was spent waiting on data from memory. This is further supported by the result that the CPU time was reduced by substantially less than the wall clock time (see Table II). In the new test, the AVX512 and AVX tests on smeagle and cheeta showed substantial improvement over using straight c code with SSE instructions allowed.

The new tests using no AVX instructions on cheeta (gcc 4.8.5) showed very poor results whereas the comparable test on smeagle (gcc 8.3.1) were more in line with expectations. These tests were repeated to confirm the results. The poor result seems confined to the Faraday synthesis section as it took practically all of the total time rather than the more typical 30% (see Table II). The no AVX new test on smeagle also used a large fraction (70%) of the total time in the Faraday synthesis section.

The Faraday synthesis is only one component of a typical run of RMSyn; a CLEAN is generally also done on each pixel with detectable polarization. While the details of the CLEAN depend on the particular case being analyzed, the test case used here is relatively representative. The fraction of the total run time used by the Faraday synthesis in the various test cases is given in Table II. The cost of the Faraday synthesis dominated all cases with the “old” implementation but is (generally) a relatively smaller component in the “new” implementation.

A portion of the field used for the timing tests in Stokes I is shown in Figure 3. Polarized AGN and extended cluster emission appears in this area. The 3D Faraday depth cube is collapsed to 2D dimensions in Figure 4.

TABLE II
RMSYNTH RUN FRACTION

System	Test	Vector	compiler	Old Fract %	New Fract %
Gandalf	Small	AVX	gcc4.8.5	77.3	30.7
Gandalf	Large	AVX	”	77.2	29.9
Cheeta	Small	AVX	gcc4.8.5	90.0	41.9
Cheeta	Large	AVX	”	93.6	63.0
Cheeta	Small	no AVX	gcc4.8.5	89.4	91.6
Cheeta	Large	no AVX	”	93.9	95.1
Smeagle	Small	AVX512	gcc8.3.1	77.0	30.2
Smeagle	Large	AVX512	”	84.5	33.8
Smeagle	Small	AVX512	icc	79.6	30.0
Smeagle	Large	AVX512	”	85.5	37.9
Smeagle	Small	AVX	gcc8.3.1	70.7	21.0
Smeagle	Large	AVX	”	79.3	23.2
Smeagle	Small	AVX	gcc4.8.5	64.1	30.4
Smeagle	Large	AVX	”	82.9	37.9
Smeagle	Small	AVX	icc	70.7	21.7
Smeagle	Large	AVX	”	79.3	31.2
Smeagle	Small	no AVX	gcc8.3.1	77.5	65.9
Smeagle	Large	no AVX	”	84.2	69.7

VI. FARADAY ANALYSIS

The previous sections have described the generation of full Faraday depth cubes. In many practical cases, the Faraday screen is relatively simple and an analysis searching for the peak Faraday depth (called “Rotation measure”) and the associated unwrapped polarized intensity and polarization angle are sufficient. There are two generic approaches to this, 1) doing a direct search, i.e. computing the full cube and in each pixel using the rotation measure and polarization corresponding to the peak unwrapped polarized amplitude. The other approach, 2) is to perform a least squares fitting to the Q and U values in each pixel. In this latter case, the fitting needs

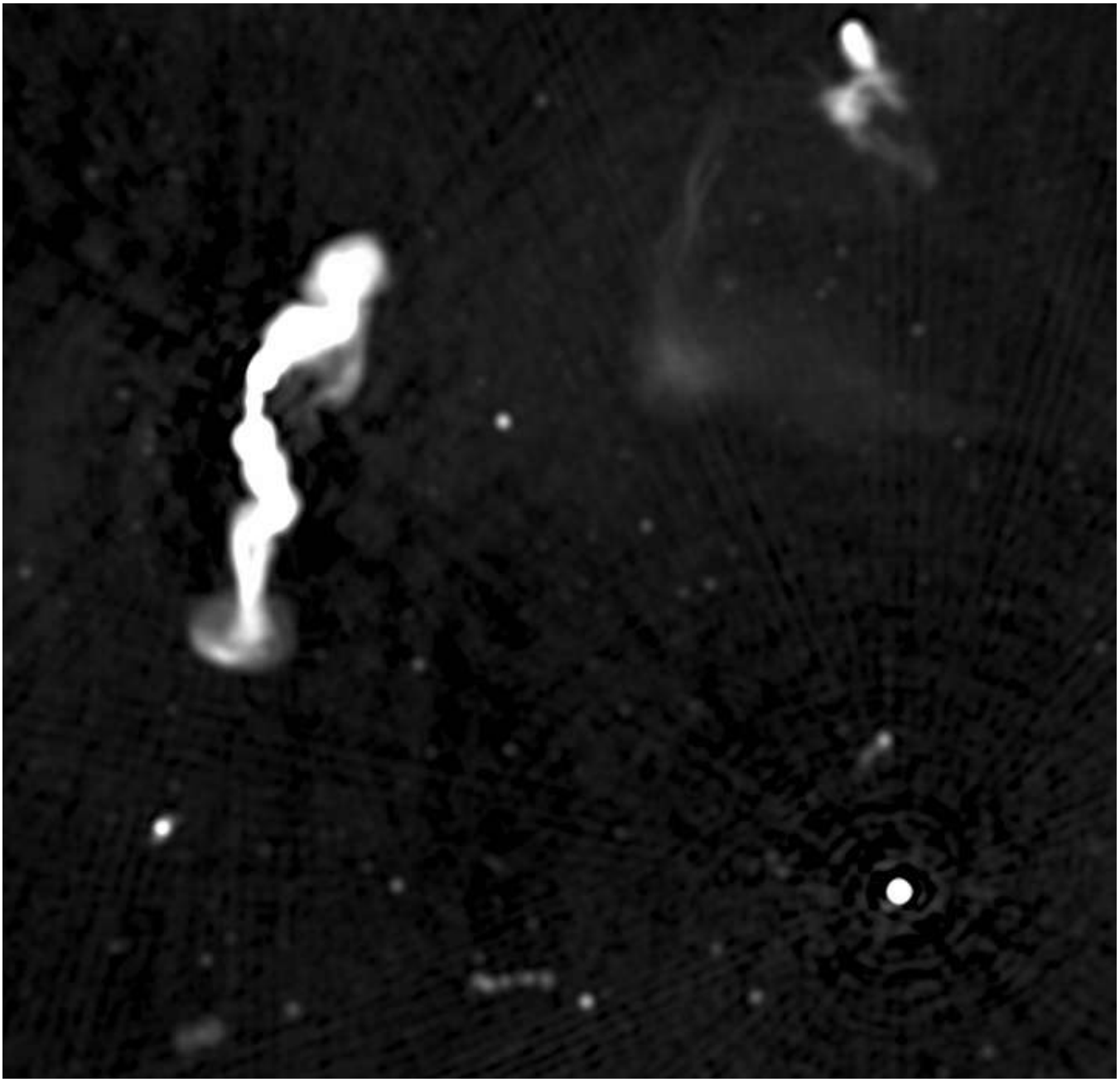

Fig. 3. Region of cluster of galaxies Abell 3395 in Stokes I at 1.3 GHz containing interesting polarized emission. From [11].

to be nonlinear as the fitted values, atan2(U,Q), are ambiguous and a good initial starting solution is needed. Such a starting solution can be obtained via a direct search. These functions have been implemented in Obit class ObitRMFit with a python interface in class RMFit as functions Cube and ImArr for Q/U cubes or arrays of single plane images.

A new task, Farad, has been implemented to use the new, higher efficiency ObitFarSyn class. The original class ObitRMFit was implemented only through a python interface which makes timing more problematic. For the timing tests a c program was used for the testing. The tests used the same three systems, gandalf, smeagle and cheeta used for the Faraday synthesis tests. The dataset was the same as the "small" ($2001\times2001\times200$) test and all runs used the best vector set available on the system. The timing results are shown in Table III.

The "RMSyn" test was looking for the peak unwrapped polarized intensity in the Faraday depth cube used the same software as for the Faraday synthesis except instead of saving the whole cube, only kept track of the peak. The least squares fitting ("LSQ") in both the old and new tests used a Gnu Scientific Library (GSL) nonlinear fitting routine. This was provided with functions to calculate the residuals and derivatives of the model. To perform this fitting, the routine had to extract all the Q and U measurements in each pixel. The two enhancements were 1) to have the threads use successive cells in the arrays rather than separated blocks of cells. This improves the cache hit ratio as data are read from memory to

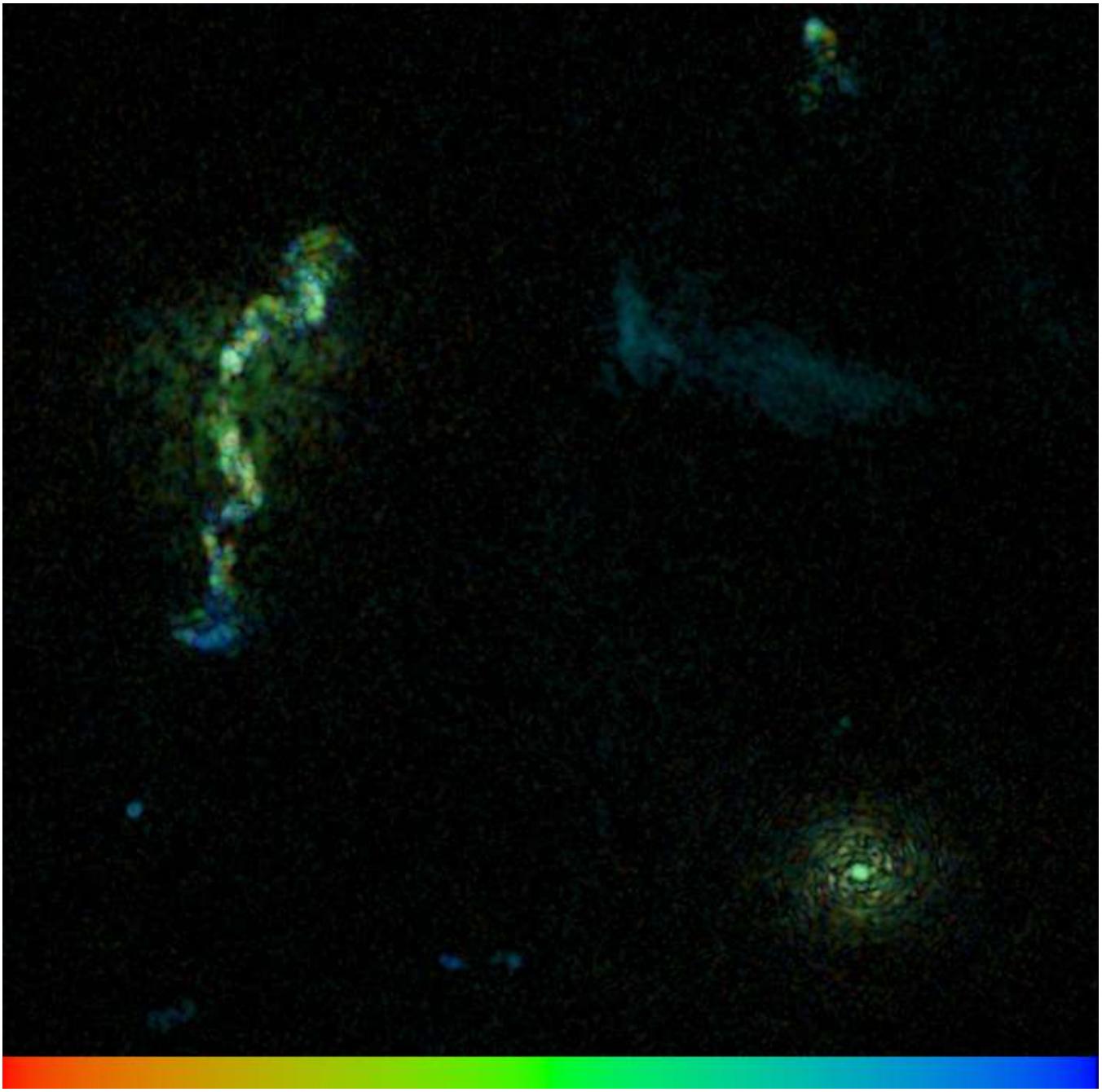

Fig. 4. False color representation of the Abell 3395 Faraday cube. This is the polarized intensity weighted Faraday depth given in color shown by the scale bar; red is -100 rad m^{-2} and blue is 100 rad m^{-2}. Colors are washed out in pixels with a range of Faraday depths. This is the same region of the sky as in Figure 3.

cache in blocks the size of the width of the memory bus; 8 or 16 floats in the tests performed.

The other enhancement, 2) was in the loops in the routines to calculate model residuals and derivatives. The interface to the GSL routines involved function calls to set each of these values; initially this was done in a single loop which tested if the weight of each datum was positive. The enhanced version replaced this with two loops, the first to calculate arrays of the values to be fed to GSL and the second to set them. This allows the compiler to make better use of optimization and vectorization. Also the test on the weight was dropped as with zero weight, the results were the same. This also allows better compiler optimization.

The results in Table III shows a modest improvement in run time for most of the tests but generally a factor of 2 reduction in the CPU time used. The timing in these tests was for the full operation and included input and output I/O.

The result of the LSQ test on gandalf is shown in Figure 5. Since only the Faraday depth of the peak unwrapped polarized intensity is shown, there is no washing out of the colors. However, this figure does not indicate areas with complex Faraday depth spectra.

TABLE III
FARADAY ANALYSIS CPU TIMING

System	Test	Old Real min.	Old CPU min.	New Real min.	New CPU min.	Real Ratio	CPU Ratio
Gandalf	RMSyn	9.6	128.4	5.1	46.5	1.9	2.8
Gandalf	LSQ	9.4	127.9	6.9	66.6	1.4	1.9
Cheeta	RMSyn	3.9	228.4	3.0	117.5	1.3	1.9
Cheeta	LSQ	4.5	263.4	4.5	150.9	1.0	1.7
Smeagle	RMSyn	4.0	78.9	3.1	38.2	1.3	2.1
Smeagle	LSQ	7.7	158.7	4.1	56.5	1.9	2.8

VII. DISCUSSION

The changes made to Faraday synthesis lead to factors of 7 to 10 reduction of the run time on the systems tested. This is largely due to the single rather than multiple passes through the input data for each output plane and dropping the testing for blanked pixels. The memory access pattern leads to a much improved cache hit ratio.

Hand coded use of vector intrinsics give much better performance than simple gcc compiler optimization. With the use of vector intrinsics the more expensive intel c compiler, icc, shows little improvement over gcc. AVX512 (16 floats) on smeagle doesn't give much enhancement over AVX (8 floats), this likely means that the memory access time is the dominant cost, 512 bits are fetched from memory at a time and if that dominates, then whether one or two floating operations is needed is irrelevant.

The new implementation of Faraday analysis in task Farad shows modest improvement in the run times but of order a factor of two in CPU usage. This suggests improved runtime performance on systems with fewer cores than used in these tests.

The weighting used in these tests were per plane but should really be done per pixel for wide fields of view for single pointing images where the relative antenna gain varies with frequency, hence with λ^2. In practice, off–axis instrumental problems such as instrumental polarization become important before this becomes an issue. This is especially true for MeerKAT. Mosaic images should have much reduced off–axis instrumental problems and no issue with frequency variable antenna gain.

REFERENCES

[1] W. D. Cotton, "Obit: A Development Environment for Astronomical Algorithms," *PASP*, vol. 120, pp. 439–448, 2008.

[2] W. D. Cotton, "Note on the Efficacy of Multi-threading in Obit," *Obit Development Memo Series*, vol. 1, pp. 1–8, 2008. [Online]. Available: https://www.cv.nrao.edu/∼bcotton/ObitDoc/Thread.pdf

[3] ——, "Notes on icc and AVX," *Obit Development Memo Series*, vol. 61, pp. 1–2, 2019. [Online]. Available: https://www.cv.nrao.edu/∼bcotton/ObitDoc/ICCAVX.pdf

[4] ——, "AVX512: First Look," *Obit Development Memo Series*, vol. 67, pp. 1–5, 2020. [Online]. Available: https://www.cv.nrao.edu/∼bcotton/ObitDoc/AVX512.pdf

[5] M. A. Brentjens and A. de Bruyn, "Faraday rotation measure synthesis," *A&A*, vol. 441, pp. 1217–+, Sep. 2005.

[6] B. J. Burn, "On the depolarization of discrete radio sources by Faraday dispersion," *MNRAS*, vol. 133, p. 67, Jan. 1966.

[7] G. Heald, R. Braun, and R. Edmonds, "The Westerbork SINGS survey. II Polarization, Faraday rotation, and magnetic fields," *A&A*, vol. 503, no. 2, pp. 409–435, Aug. 2009.

[8] X. H. Sun, L. Rudnick, T. Akahori, C. S. Anderson, M. R. Bell, J. D. Bray, J. S. Farnes, S. Ideguchi, K. Kumazaki, T. O'Brien, S. P. O'Sullivan, A. M. M. Scaife, R. Stepanov, J. Stil, K. Takahashi, R. J. van Weeren, and M. Wolleben, "Comparison of Algorithms for Determination of Rotation Measure and Faraday Structure. I. 1100-1400 MHz," *AJ*, vol. 149, no. 2, p. 60, Feb. 2015.

[9] L. Rudnick and W. D. Cotton, "Correction to: Full resolution deconvolution of complex Faraday spectra," *MNRAS*, vol. 523, no. 1, pp. 774–774, Jul. 2023.

[10] W. D. Cotton and L. Rudnick, "Faraday Synthesis of Unequally Spaced Data and Complex Deconvolution," *Obit Development Memo Series*, vol. 76, pp. 1–18, 2022. [Online]. Available: https://www.cv.nrao.edu/∼bcotton/Obitdoc/RMSyn.pdf

[11] K. Knowles, W. D. Cotton, L. Rudnick, F. Camilo, S. Goedhart, R. Deane, M. Ramatsoku, M. F. Bietenholz, M. Brüggen, C. Button, H. Chen, J. O. Chibueze, T. E. Clarke, F. de Gasperin, R. Ianjamasimanana, G. I. G. Józsa, M. Hilton, K. C. Kesebonye, K. Kolokythas, R. C. Kraan-Korteweg, G. Lawrie, M. Lochner, S. I. Loubser, P. Marchegiani, N. Mhlahlo, K. Moodley, E. Murphy, B. Namumba, N. Oozeer, V. Parekh, D. S. Pillay, S. S. Passmoor, A. J. T. Ramaila, S. Ranchod, E. Retana-Montenegro, L. Sebokolodi, S. P. Sikhosana, O. Smirnov, K. Thorat, T. Venturi, T. D. Abbott, R. M. Adam, G. Adams, M. A. Aldera, E. F. Bauermeister, T. G. H. Bennett, W. A. Bode, D. H. Botha, A. G. Botha, L. R. S. Brederode, S. Buchner, J. P. Burger, T. Cheetham, D. I. L. de Villiers, M. A. Dikgale-Mahlakoana, L. J. du Toit, S. W. P. Esterhuyse, G. Fadana, B. L. Fanaroff, S. Fataar, A. R. Foley, D. J. Fourie, B. S. Frank, R. R. G. Gamatham, T. G. Gatsi, M. Geyer, M. Gouws, S. C. Gumede, I. Heywood, M. J. Hlakola, A. Hokwana, S. W. Hoosen, D. M. Horn, J. M. G. Horrell, B. V. Hugo, A. R. Isaacson, J. L. Jonas, J. D. B. Jordaan, A. F. Joubert, R. P. M. Julie, F. B. Kapp, V. A. Kasper, J. S. Kenyon, P. P. A. Kotzé, A. G. Kotze, N. Kriek, H. Kriel, V. K. Krishnan, T. W. Kusel, L. S. Legodi, R. Lehmensiek, D. Liebenberg, R. T. Lord, B. M. Lunsky, K. Madisa, L. G. Magnus, J. P. L. Main, A. Makhaba, S. Makhathini, J. A. Malan, J. R. Manley, S. J. Marais, M. D. J. Maree, A. Martens, T. Mauch, K. McAlpine, B. C. Merry, R. P. Millenaar, O. J. Mokone, T. E. Monama, M. C. Mphego, W. S. New, B. Ngcebetsha, K. J. Ngoasheng, M. T. Ockards, A. J. Otto, A. A. Patel, A. Peens-Hough, S. J. Perkins, N. M. Ramanujam, Z. R. Ramudzuli, S. M. Ratcliffe, R. Renil, A. Robyntjies, A. N. Rust, S. Salie, N. Sambu, C. T. G. Schollar, L. C. Schwardt, R. L. Schwartz, M. Serylak, R. Siebrits, S. K. Sirothia, M. Slabber, L. Sofeya, B. Taljaard, C. Tasse, A. J. Tiplady, O. Toruvanda, S. N. Twum, T. J. van Balla, A. van der Byl, C. van der Merwe, C. L. van Dyk, V. Van Tonder, R. Van Wyk, A. J. Venter, M. Venter, M. G. Welz, L. P. Williams, and B. Xaia, "The MeerKAT Galaxy Cluster Legacy Survey. I. Survey Overview and Highlights," *A&A*, vol. 657, p. A56, Jan. 2022.

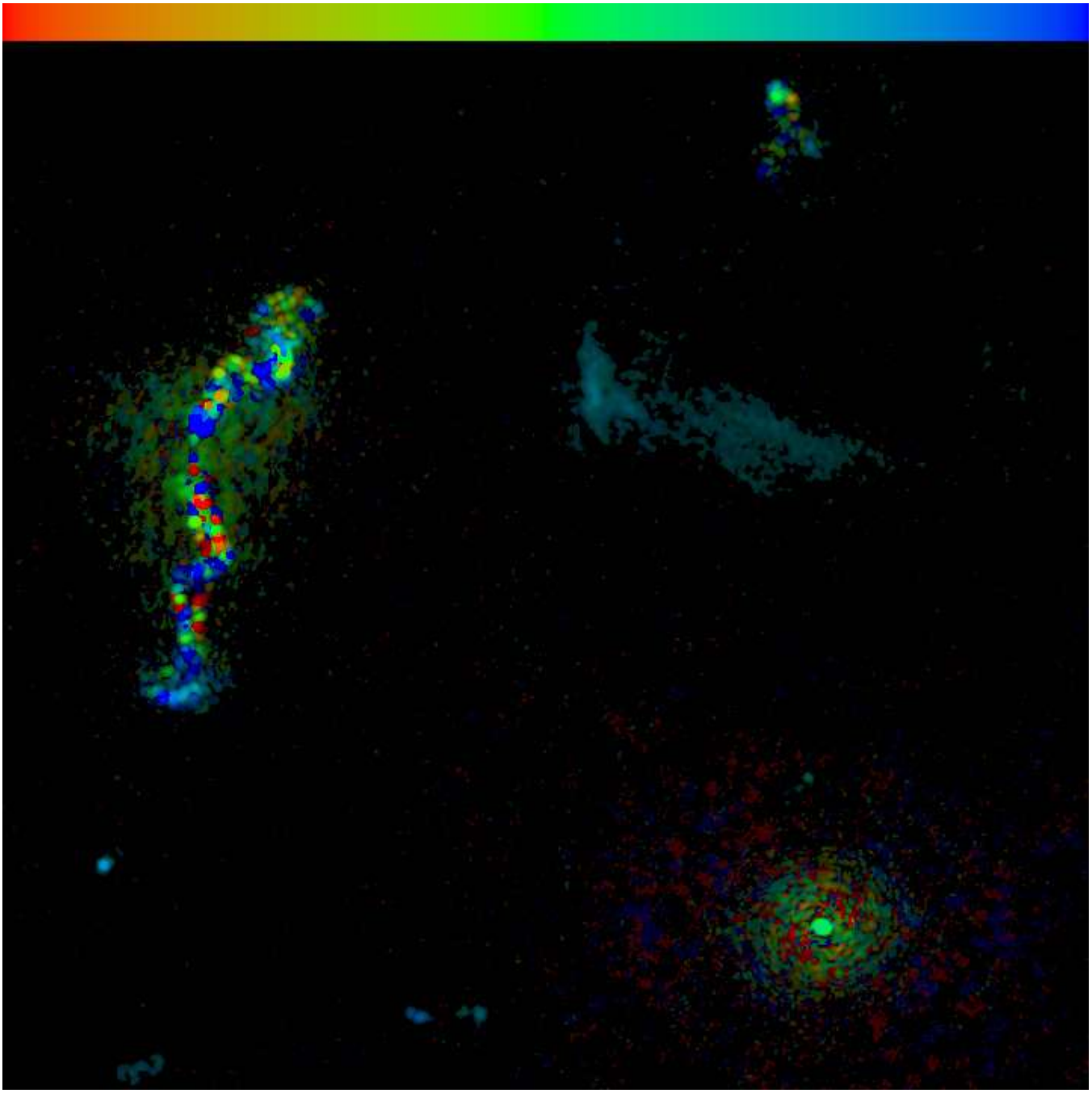

Fig. 5. Hue-Intensity image of Abell 3395 derived from a Faraday analysis. The intensity is the unwrapped polarized intensity and the color is the peak rotation measure given by the scale bar; red is -100 rad m^{-2} and blue is 100 rad m^{-2}. This is the same region of the sky as in Figure 3.