Non-thermal emission and magnetic fields in nearby galaxies
Seethapuram Sridhar, Sarrvesh

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2018

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 02-11-2019
Chapter 5

cuFFS: A GPU-accelerated Rotation Measure Synthesis Code

Sridhar, S. S., Heald, G., & van de Hulst, J. M.
Under review with Astronomy & Computing

121
5.1 Introduction

A processor is the logic circuit that responds to and processes the basic instructions (fetch, decode, execute, writeback) that drive a computer. Owing to improvements in semiconductor technology, processor manufacturers have been able to pack more transistors onto a single chip while also increasing the processor clock rate\(^1\) thereby achieving higher processing speeds. For example, Intel’s 4004 processor released in 1971 had a clock rate of 108 kHz while the Intel Core i7 which is currently available in the market has a clock rate of about 4.2 GHz. Until the early 2000s, improvements in the execution speeds of scientific programs were largely due to the increase in CPU clock rates. This however changed in last decade when chip manufacturers realised that it was hard to keep increasing the clock rate as it led to overheating of the chips due to thermal losses (Markoff 2004).

Processor manufacturers work around the “thermal wall” by implementing multiple processing units (or cores) on a single chip leading to the design of multicore processors. While each core on the chip is not faster than its predecessor, multicore processors achieve higher compute throughput using parallelisation. Processor vendors typically ship commercially available chips with upto 32 cores. GPUs extend this strategy of implementing multiple cores on a single chip to an extreme limit. In comparison to multicore CPUs, GPUs achieve high compute throughput by maximising the number of processing units (or cores) on a chip. For example, NVIDIA’s Tesla P100 card has 56 streaming microprocessors each with 64 processing units resulting in a total of 3584 single precision (or 1792 double precision) floating point compute cores on a single chip. An added advantage of using GPUs over CPU compute clusters is that GPUs consume less energy measured in Watts per FLOP\(^2\) (Trancoso & Charalambous 2005; Michalakes & Vachharajani 2008).

While GPUs were originally designed to be used as graphics coprocessors\(^3\) to handle higher resolution and display rates demanded by the gaming industry, the massive compute capability of GPUs has been exploited leading to the creation of a new field called GPGPU (General Purpose GPU) computing (Owens et al. 2007). While the initial scientific programs that were run on GPUs had to be ported to the GPU native programming languages, software solutions made available in the last decade have made it far easier to develop GPU codes (Nvidia 2017). A review of GPU use in scientific computing can be found in Owens et al. (2007) and Owens et al. (2008).

Graphical processing units have been used quite extensively in astronomy in areas of research ranging from real-time applications like adaptive optics (for example, see Bouchez et al. 2012) and correlators for radio interferometry (Schaaf & Overeem 2004), easily parallelisable algorithms like N-body simulations (Portegies Zwart et al. 2007; Elsen et al. 2007), to more complex algorithms in

---

\(^1\)The clock rate of a processor is proportional to the number of instructions that a processor can execute per unit time.

\(^2\)Floating point operations

\(^3\)Coprocessors are secondary processors mounted on a device that complements the compute capabilities of the primary processor.
general relativistic magnetohydrodynamics (Zink 2011). For a detailed discussion on the use of GPUs in astronomy, see Fluke et al. (2011) and Fluke (2012).

Within the field of radio astronomy, GPUs have largely been restricted to implementing fast real-time signal processing applications like correlators and real-time calibration pipelines (for example, see Schaaf & Overeem 2004; Harris et al. 2008; Ord et al. 2009; Chennamangalam et al. 2014; Price et al. 2016), and pulsar de-dispersion pipelines (Magro et al. 2011). Recently, the Dynamic Radio Astronomy of Galactic Neutron stars and Extragalactic Transients (DRAGNET; Bassa et al. 2017) project with the LOFAR radio telescope has started using GPUs to search for fast radio transients in real-time. GPUs have also been used to accelerate solving the Radio Interferometer Measurement Equation (Hamaker et al. 1996) to calibrate radio interferometry data (Perkins et al. 2015). Within the context of medical imaging, it has been demonstrated that the gridding procedure that needs to be carried out before applying two-dimensional inverse Fourier transform to the data can be accelerated using GPUs (Schomberg & Timmer 1995; Schiwietz et al. 2006). Furthermore, a GPU implementation of the fast Fourier transform library cuFFT is available as part of the official CUDA toolkit.

In this chapter, we present a GPU-accelerated program – CUDA-accelerated Fast Faraday Synthesis (cuFFS) – to perform a commonly used technique in radio polarimetry called Faraday rotation measure (RM) synthesis. This chapter is organised as follows: The theoretical background and computational complexity of the rotation measure synthesis method are presented in §5.2.1 and 5.2.2. Our GPU implementation of the RM Synthesis code is explained in §5.3 and science verification and benchmark tests are presented in section §5.3.2. Finally, we present our conclusion and future prospects for improvements to the software package in §5.4.

5.2 Background

5.2.1 RM synthesis: Theory

Linearly polarized synchrotron emission is an important observational probe to study astrophysical magnetic fields. Synchrotron emission is produced by cosmic ray electrons accelerating across magnetic field lines. The polarization angle of the synchrotron radiation depends on the orientation of the magnetic field in the plane of the sky. Thus resolved polarimetric observations allow us to probe and study the structure of astrophysical magnetic field\(^5\). However, in practice, this is complicated owing to the fact that the observed polarization angle and polarization fraction\(^6\) is different from the intrinsic properties at the source due to propagation effects caused by Faraday rotation.

\(^4\text{www.astron.nl/dragnet/}\)

\(^5\)To be precise, polarized synchrotron emission probes only the ordered component of the magnetic field. Distinction between different components of astrophysical magnetic fields and the observational probes used to study them is beyond the scope of this chapter. For a recent review on this topic, see Fletcher & Klein (2015) and Beck (2016).

\(^6\)polarized intensity divided by total intensity of the synchrotron radiation
Faraday rotation is a phenomenon by which the plane of polarization of a linearly polarized electromagnetic vector gets rotated due to circular birefringence as it propagates through a magneto-ionic medium like the interstellar medium or the ionosphere. The observed polarization angle ($\chi$) is related to the intrinsic polarization angle ($\chi_0$) through a quantity called Faraday depth ($\phi$) as

$$\chi(\lambda^2) = \chi_0 + \phi\lambda^2$$  \hspace{1cm} (5.1)

where $\phi$ is related to the strength of the magnetic field projected along the line of sight ($B_\parallel = \vec{B} \cdot \vec{d}$) as

$$\phi = 0.81 \int_{\text{observer}}^{\text{source}} n_e \vec{B} \cdot d\vec{l}.$$  \hspace{1cm} (5.2)

In the above equation, $n_e$ is the number density of electrons measured in units of cm$^{-2}$, $l$ is the pathlength in parsec, $B$ is in $\mu$G, and $\phi$ is in rad m$^{-2}$.

The observed complex polarization vector $P(\lambda^2) = Q(\lambda^2) + iU(\lambda^2)$ is related to the observed polarization angle $\chi(\lambda^2)$, fractional polarization $p$, and total intensity $I$ as

$$P(\lambda^2) = pI e^{2i\chi(\lambda^2)}.$$  \hspace{1cm} (5.3)

Note that equations 5.1 and 5.3 allow us to relate the observed polarization vector with the intrinsic polarization angle and the Faraday rotation measure. Replacing the Faraday rotation measure with a generalized Faraday depth $\phi$, and since emission from multiple $\phi$ can contribute to the observed polarization vector $P(\lambda^2)$, we can write

$$P(\lambda^2) = \int_{-\infty}^{+\infty} pI e^{2i[\chi_0 + \phi\lambda^2]} d\phi = \int_{-\infty}^{+\infty} F(\phi) e^{2i\phi\lambda^2} d\phi.$$  \hspace{1cm} (5.4)

where $F(\phi)$ is defined as the Faraday Dispersion Function $F(\phi) := pI e^{2i\chi_0}$. Inverting this Fourier transform-like equation,

$$F(\phi) = \int_{-\infty}^{+\infty} P(\lambda^2) e^{-2i\phi\lambda^2} d\lambda^2$$  \hspace{1cm} (5.5)

we can see that the Faraday Dispersion Function (FDF) represents the emitted polarized flux as a function of Faraday depth. From equations 5.4 and 5.5, it is easy to see that $P(\lambda^2)$ and $F(\phi)$ form a Fourier-like transform pair.

Recovering $F(\phi)$ from the observed $P(\lambda^2)$ is however not straight-forward as we cannot measure $P(\lambda^2)$ for $\lambda < 0$. Furthermore, $P(\lambda^2)$ is also not measured for all possible values of $\lambda > 0$. Rotation measure synthesis is a technique that was formulated by Brentjens & de Bruyn (2005) which tries to recover $F(\phi)$ using a discrete sampling of $P(\lambda^2)$. In addition to reconstructing $F(\phi)$, the procedure also improves the sensitivity of the polarization measurement by adding up derotated polarization signal across the entire observed bandwidth. In the following section, we provide a brief overview of the procedure. For a detailed account on RM synthesis, we refer the reader to Brentjens & de Bruyn (2005) and Heald (2009).
The rotation measure synthesis technique works around this problem by introducing a window function $W(\lambda^2)$ which is non-zero only for $\lambda^2$ values at which complex polarization $P(\lambda^2)$ has been measured. Thus, equation 5.4 can be modified as

$$\tilde{P}(\lambda^2) = W(\lambda^2)P(\lambda^2) = W(\lambda^2) \int_{-\infty}^{+\infty} P(\lambda^2)e^{-2i\phi\lambda^2} d\lambda^2$$  \hspace{1cm} (5.6)$$

Combining the above equation with equation 5.5 and applying the convolution theorem, the reconstructed Faraday dispersion function $\tilde{F}(\phi)$ is related to the “true” Faraday dispersion function as

$$\tilde{F}(\phi) = F(\phi) * R(\phi) = K \int_{-\infty}^{+\infty} \tilde{P}(\lambda^2)e^{-2i\phi\lambda^2} d\lambda^2$$  \hspace{1cm} (5.7)$$

From the above equation, it is easy to see that the recovered Faraday dispersion function is a convolution of the “true” Faraday dispersion function with smoothing kernel called the Rotation Measure Spread Function (RMSF). Conceptually, this is analogous to synthesis imaging where $\tilde{F}(\phi)$ is equivalent to the dirty image and $R(\phi)$ is equivalent to the dirty beam. The RMSF is related to the weight function $W(\lambda^2)$ as

$$R(\phi) = K \int_{-\infty}^{+\infty} W(\lambda^2)e^{-2i\phi\lambda^2} d\lambda^2$$  \hspace{1cm} (5.8)$$

where

$$K = \left( \int_{-\infty}^{+\infty} W(\lambda^2)d\lambda^2 \right)^{-1}.$$  \hspace{1cm} (5.9)$$

5.2.2 RM synthesis: In practice

As discussed in the previous section, the complex polarization vector is not measured for all values of $\lambda^2$. Modern radio telescopes measure $P(\lambda^2)$ in narrow frequency channels spread across a wide bandwidth each with equal bandwidth. In such a scenario, the integrals in equations 5.7, 5.8 and 5.9 can be replaced by sums over each frequency channel provided $\phi\delta\lambda^2 \ll 1:

$$\tilde{F}(\phi) \approx K \sum_{i=1}^{N} \tilde{P}_i e^{-2i\phi(\lambda_i^2-\lambda_0^2)}$$  \hspace{1cm} (5.10)$$

$$R(\phi) \approx K \sum_{i=1}^{N} w_i e^{-2i\phi(\lambda_i^2-\lambda_0^2)}$$  \hspace{1cm} (5.11)$$

$$K = \left( \sum_{i=1}^{N} w_i \right)^{-1}.$$  \hspace{1cm} (5.12)$$

In the above equations, $\tilde{P}_i$ is the complex polarization vector measured at $\lambda_i$ and $w_i = W(\lambda_i)$. Figure 5.1 shows a RMSF calculated using equation 5.11 for
a mock L-band dataset consisting of 1000 frequency channels covering the 1 – 2 GHz frequency range. Notice in equations 5.10 and 5.11, we have introduced an additional term $\lambda_0^2$ in the exponent which is set to the mean of $\lambda_i^2$. Introducing this additional $\lambda_0^2$ term is useful as it prevents rapid rotation of $\tilde{Q}(\phi)$ and $\tilde{U}(\phi)$ as can be seen in Figure 5.1. Scaling these two quantities allows one to make an accurate estimate of $Q_{\text{peak}}$ and $U_{\text{peak}}$ and thus make an accurate estimate of the polarization angle $\chi = 0.5 \tan^{-1}(U/Q)$.

The aim of the RM synthesis software package is to compute equations 5.10 and 5.12. Equation 5.11 is needed only if one wishes to deconvolve $R(\phi)$ from the reconstructed Faraday dispersion function using a procedure called RM-CLEAN\textsuperscript{7} (Heald 2009).

Since $\tilde{F}(\phi)$ and $\tilde{P}(\phi)$ in equation 5.10 are complex vectors, the equation can be written in terms of its Stokes $Q$ and Stokes $U$ components as

$$\tilde{Q}(\phi) = K \sum_{i=1}^{N} Q_{\lambda_i} \cos 2\phi(\lambda_i^2 - \lambda_0^2) + U_{\lambda_i} \sin 2\phi(\lambda_i^2 - \lambda_0^2)$$  \hspace{1cm} (5.13)

$$\tilde{U}(\phi) = K \sum_{i=1}^{N} U_{\lambda_i} \cos 2\phi(\lambda_i^2 - \lambda_0^2) - Q_{\lambda_i} \sin 2\phi(\lambda_i^2 - \lambda_0^2).$$  \hspace{1cm} (5.14)

$$\tilde{P}(\phi) = \sqrt{\tilde{Q}^2(\phi) + \tilde{U}^2(\phi)}$$  \hspace{1cm} (5.15)

The above two equations compute the recovered Faraday dispersion function for a given value of $\phi$. In practice, the above equations need to be computed for a wide range of values of $\phi$ to get a sense for the variation of the intrinsic polarization vector as a function of Faraday depth. Thus the above equations become

$$\tilde{Q}(\phi_j) = K \sum_{i=1}^{N} Q_{\lambda_i} \cos 2\phi_j(\lambda_i^2 - \lambda_0^2) + U_{\lambda_i} \sin 2\phi_j(\lambda_i^2 - \lambda_0^2); \forall \phi_j \in [\phi_{\text{min}}, \phi_{\text{max}}]$$  \hspace{1cm} (5.16)

$$\tilde{U}(\phi_j) = K \sum_{i=1}^{N} U_{\lambda_i} \cos 2\phi_j(\lambda_i^2 - \lambda_0^2) - Q_{\lambda_i} \sin 2\phi_j(\lambda_i^2 - \lambda_0^2); \forall \phi_j \in [\phi_{\text{min}}, \phi_{\text{max}}].$$  \hspace{1cm} (5.17)

These two equations need to be computed for each line of sight in a radio image to construct a 3D RM-cube contains the Faraday dispersion function for all lines of sight.

### 5.2.3 RM synthesis: Computational costs

From equations 5.13 and 5.14, it is easy to see that $K$, $2\phi$, and $\lambda_i^2 - \lambda_0^2$ need to computed only once for the entire dataset and have negligible compute cost.

\textsuperscript{7}RM-CLEAN is not supported in the current version of cuFFS. However, we intend to implement this in the next code release.
Figure 5.1 – Rotation Measure Spread Function (RMSF) for a mock L-band dataset consisting of 1000 frequency channels covering the 1 – 2 GHz frequency range with $\lambda_0^2 = 0$ (top) and $\lambda_0^2 \neq 0$ (bottom).
The compute cost to calculate $\tilde{Q}(\phi)$, $\tilde{U}(\phi)$, and $\tilde{P}(\phi)$ for one $\phi$ and one $\lambda^2$ is 13 FLOP (7 multiplications, 2 trigonometric functions, 2 additions, 1 subtraction and 1 square root operation).

Thus to calculate $\tilde{Q}(\phi)$, $\tilde{U}(\phi)$, and $\tilde{P}(\phi)$ for one Faraday depth value using $N_{\text{chan}}$ frequency channels is

- $15 \times N_{\text{chan}}$ FLOP to compute equations 5.13 – 5.15 for a given $\phi$ using all $\lambda_i^2$.
- $2(N_{\text{chan}} - 1)$ additions for summations over all $i$.
- 2 multiplications to scale the values by $K$.

The above operations need to be repeated $N_{\phi}$ times to create RM-cubes of $\tilde{Q}(\phi)$, $\tilde{U}(\phi)$, and $\tilde{P}(\phi)$ with $N_{\phi}$ Faraday depth planes using $N_{\text{chan}}$ input frequency channels. This amount to a total computation cost of $15N_{\text{chan}}N_{\phi}$ FLOP for a single line of sight. Since each line of sight in the input datacubes can be treated independently, we can scale the computation cost further by the number of spatial pixels in the input datacubes making the final computational cost to be $\sim 15N_{\text{chan}}N_{\phi}N_{\text{los}}$.

For a typical polarimetry data set from the Westerbork Synthesis Radio Telescope, for example from the Westerbork Spitzer Nearby Galaxy Survey (SINGS) survey (Braun et al. 2007; Heald et al. 2009), which has $512^2$ spatial pixels, 803 frequency channels and 401 Faraday depth planes, the computational cost required to carry out RM synthesis is about 1.27 TFLOP$^8$.

To make this estimate, we assumed that each arithmetic and trigonometric functions amount to a single floating point operation. In practice, this is however not true. For example, standard libraries evaluate trigonometric functions as Taylor series expansions and the number of Taylor terms used in the expansion can be implementation-dependent. Furthermore, the exact number of instructions needed for different mathematical operations available in the standard C libraries can vary depending on the implementation, system hardware and compiler optimisation options. Thus the above compute cost should be treated only as a lower limit and not as a precise value.

### 5.3 GPU implementation of RM Synthesis

cuFFS is written in C, and GPU acceleration is achieved using the CUDA Application Programming Interface (API). User input is provided to the program using a configuration file which is parsed using the structured configuration file parser libconfih$^9$. Sample input parsel file and output produced by the program are shown in Figures 5.2 and 5.3.

Input and output data cubes can either be in FITS (Pence 1999) or in HDFITS (Price et al. 2015) formats. FITS files are read in and written out using the

---

$^8$1 TFLOP = 1 trillion floating-point operations.

$^9$http://www.hyperrealm.com/libconfig/
5.3. GPU IMPLEMENTATION OF RM SYNTHESIS

// How many GPUs to use?
// In version 1.0, this is always 1.
nGPU = 1;

// What is the input format?
// Can be "FITS" or "HDF5" (not case-sensitive).
fileFormat = "FITS";

// Define how fits files are stored on disk
qCubeName = "cube-Q.rot.fits";
uCubeName = "cube-U.rot.fits";
freqFileName = "frequencies-Q.txt";

// Define the faraday depth axis. All units in rad/m/m
plotRMSF = False;
phiMin = -500.;
nPhi = 401; // integer
dPhi = 2.5;

// Prefix for output filenames
outPrefix = "trial1_";

Figure 5.2 – Sample input parset to cuFFS.

CFITSIO\textsuperscript{10} library while the HDFITS is accessed using the HDF5 library\textsuperscript{11}. Justification for supporting data formats in addition to the standard FITS format is provided in section 5.3.1.

A fast CPU implementation of RM synthesis, like Brentjens (2007), computes the RM cubes by evaluating all Faraday depth values for each input channel. The main advantage of using such a strategy is that it minimises disk I/O as each input channel is read only once and the output cubes are written to disk only once as long as the output cubes fit in the memory. If all output Faraday depth planes do not fit in memory, the CPU program computes a subset of Faraday depth planes that fits in memory, writes the output planes to disk, and then proceeds to compute the next subset of Faraday depth planes.

Note that adopting a similar scheme is not very efficient on a GPU as this CPU implementation would involve numerous data transfers between the host and the device. To speed up RM synthesis using a GPU, a new programming model that reduces the data transfer between the host and the GPU is needed. Since each line of sight in the data cube can be processed independent of each other, each core on the processor can be assigned to process a particular line of sight and hence RM synthesis can be efficiently implemented on multi-core architectures like GPUs.

\textsuperscript{10}https://heasarc.gsfc.nasa.gov/fitsio/
\textsuperscript{11}https://support.hdfgroup.org/HDF5/
RM Synthesis v0.2
Written by Sarrvesh S. Sridhar
INFO: Parsing input file /home/see041/RMSynth_GPU/parsetFile
INFO: Checking input files
INFO: Detected 1 CUDA-supported GPU(s)
INFO: Selected device 0

####################################
Q Cube: /data/see041/lofar_lowres/fits/cube-U.rot.fits
U Cube: /data/see041/lofar_lowres/fits/cube-U.rot.fits
phi min: -350.00
# of phi planes: 7000
delta phi: 0.10

Input dimension: 540 x 540 x 850
Output dimension: 540 x 540 x 7000
####################################
INFO: Computing RMSF
INFO: Writing RMSF to trial1rmsf.txt
INFO: Starting RM Synthesis
INFO: Launching 540x219 blocks each with 32 threads
INFO: Timing Information
  Input read time: 0.647 s
  Compute time: 0.386 s
  Output write time: 12.966 s
  D2H Transfer time: 0.219 s
INFO: Total execution time: 0:0:17

Figure 5.3 – Sample output produced by cuFFS.
5.3. GPU IMPLEMENTATION OF RM SYNTHESIS

Figure 5.4 shows a flow chart representation of the program execution cycle. Transferring data between the host (CPU) and the device (GPU) can be expensive and time-consuming. To maximise the compute-to-transfer time ratio, our implementation ensures that the same data is transferred between the host and the device only once. The program carries out the following steps for each line of sight:

1. Transfer $\tilde{Q}(\lambda^2)$ and $\tilde{U}(\lambda^2)$ to the GPU memory.
2. Each thread on the GPU computes $\tilde{Q}(\phi)$, $\tilde{U}(\phi)$, and $\tilde{P}(\phi)$ for a given Faraday depth value.
3. Transfer the computed $\tilde{Q}(\phi)$, $\tilde{U}(\phi)$, and $\tilde{P}(\phi)$ to the host memory.

By repeating these steps for each line of sight in the input datacubes, the program constructs the final RM cubes.

Modern GPUs come with substantial amount of DRAM which implies that multiple lines of sight can be processed at the same time. Additionally, transferring a larger chunk of data between host memory and device memory is more efficient than issuing multiple data transfers. As a result, in practice, cuFFS processes multiple lines of sight at once. The GPU kernel is implemented such that each spawned thread on the GPU computes $Q_i(\phi_j)$, $U_i(\phi_j)$ and $P_i(\phi_j)$ for a given line of sight ($i$) for a fixed Faraday depth ($\phi_j$). Thus, the total number of independent threads that are executed on the GPU is $(N_\phi \times N_{\text{los}})$ where $N_{\text{los}}$ is the total number of independent sight lines processed at a time and $N_\phi$ is the number of output Faraday depth values that need to be computed.

Our code is publically available through Github\textsuperscript{12}.

5.3.1 FITS, HDF5, and HDFITS

As discussed in the previous section, our code achieves parallelism by exploiting the fact that each line-of-sight and each Faraday depth output for a given line of sight in RM synthesis can be treated independently of each other. However, during the initial development and testing phase, we quickly realised that reading and writing the FITS data cubes along the third data axis (or each line-of-sight) was the major bottleneck in our code. Even for small datacubes like the ones corresponding to WSRT observations, reading and writing datacubes along the third data axis was significantly longer than the amount of time required to carry out the actual RM synthesis computation. In addition to the bottleneck caused by the FITS I/O, it is becoming clear that the FITS data format is not best suited for large multi-dimensional datasets. An excellent review of the limitations of the FITS data format in the era of big data (radio) astronomy is presented in Thomas et al. (2015).

The slow read/write speed associated with the FITS format can be alleviated by rotating (for example using the reorder task in miriad (Sault et al. 1995)) the input datacubes such that the frequency axis is the fastest-varying dimension.

\textsuperscript{12}https://github.com/sarrvesh/RMSynth_GPU
Figure 5.4 – A flowchart depicting the execution cycle of cuFFS. All nodes shown in blue represent operations carried out on the CPU nodes shown in red indicate operations performed on the GPU.
After running cuFFS, the output datacubes, which contain the Faraday depth axis as the fastest-varying dimension, can be derotated back to the original order. This rotation can be easily implemented in python-based imaging pipelines using NumPy’s `numpy.memmap` and `numpy.swapaxis` functions. For the LOFAR HR dataset described in the next section, this rotation has an incremental overhead cost of a few seconds.

Another approach to reduce the read/write time would be to adopt data formats other than the standard FITS file format. File formats like the Hierarchical Data Format version 5 (HDF5) are well suited for storing large datacubes and have a significantly lower read/write time compared to the FITS format\(^\text{13}\) especially while reading along the slowest-varying axis. Recently, Price et al. (2015) proposed a new format called HDFITS which merges the FITS data model with the HDF5 file format. Note that data model defines how a dataset is structured and file format specifies how the bits are stored on disk. A similar strategy has also been adopted by fields like climatology and oceanography where the NetCDF data format is built on top of the HDF5 file format. In the remainder of this section, we provide a brief overview of the HDF5 file format and explain how HDFITS ports the FITS data model to the HDF5 file format.

HDF5 (Hierarchical Data Format version 5) can be thought of as a container that is capable of storing heterogeneous datasets. The basic components of an HDF5 file are groups and datasets and hierarchy within a dataset is achieved using a binary tree-like arrangement. Figure 5.5 shows the structure of a HDF5 file. In addition to groups and datasets, each HDF5 object can have one or more attributes that provide additional information about the corresponding object. This hierarchical nature of HDF5 supported by a variety of metadata makes it easy to handle complex data objects. A detailed documentation on HDF5 can be found in The HDF Group (2011).

A Single Image FITS (SIF) file consists of primary header and data unit (HDU; FITS Working Group 2008). The structure of a sample HDFITS image is shown in Listing 5.1. The HDFITS file has two groups: ROOT (indicated as “/”) and PRIMARY. The PRIMARY group, which is the child of the ROOT group, contains the image data (defined as the dataset DATA) which in this example has dimension $1 \times 1 \times 2048 \times 2048$. In addition to the dataset DATA, notice that the group PRIMARY also has attributes named CDELT1 and BMAJ. In the HDFITS format, the FITS keywords are all stored as attributes of the PRIMARY group. The main advantage of using HDFITS in our program is that the structure of the dataset stored in the HDF5 file is similar to the standard FITS format with the added advantage of higher read/write speed.

Listing 5.1 – Output of the h5dump tool showing the hierarchical layout of an HDFITS image using the Data Definition Language (DDL) syntax. For the sake of brevity, only two FITS keywords (CDELT1 and BMAJ) are shown here.

```plaintext
HDF5 "NGC1569-final.h5" {
  GROUP "/" {
    ATTRIBUTE "CLASS" {
```

\(^{13}\)Shortridge (2015) has an excellent overview of new data formats that have recently been proposed.
Figure 5.5 – Structure of a HDF5 containing the group and dataset objects arranged as a binary tree. Image adapted from The HDF Group (2011).

```
DATATYPE H5T_STRING {
  STRSIZE 10;
  STRPAD H5T_STR_NULLPAD;
  CSET H5T_CSET_ASCII;
  CTYPE H5T_C_S1;
}
DATASPACE SCALAR
GROUP "PRIMARY" {
  ATTRIBUTE "CDELT1" {
    DATATYPE H5T_IEEE_F64LE
    DATASPACE SIMPLE { ( 1 ) / ( 1 ) }
  };
  ATTRIBUTE "BMAJ" {
    DATATYPE H5T_IEEE_F64LE
    DATASPACE SIMPLE { ( 1 ) / ( 1 ) }
  }
  DATASET "DATA" {
    DATATYPE H5T_IEEE_F32BE
    DATASPACE SIMPLE { ( 1, 1, 2048, 2048 ) / ( 1, 1, 2048, 2048 ) }
    ATTRIBUTE "CLASS" {
      DATATYPE H5T_STRING {
        STRSIZE 9;
        STRPAD H5T_STR_NULLPAD;
        CSET H5T_CSET_ASCII;
        CTYPE H5T_C_S1;
      }
    }
  }
```
To achieve higher read/write throughput and to be consistent with existing data format, the current version of cuFFS supports input and output datacubes in both FITS and HDFITS formats. While operating in the FITS mode, cuFFS assumes that the user has rotated the input cubes such that the frequency axis is the fast-varying dimension. Since HDF5/HDFITS is efficient in reading along the frequency axis, no such prior rotation is required while using cuFFS in HDF5 mode.

As discussed above, numerous data formats have been proposed in the recent years for archiving large datasets. If desired, additional data formats like N-dimensional Data Format (NDF; Jenness et al. 2015), Starlink Hierarchical Data System (HDS; Jenness 2015) and Advanced Scientific Data Format (ASDF; Greenfield et al. 2015) can be supported easily with minimal change to the existing code.

5.3.2 Science verification and benchmarks

To verify the output produced by cuFFS, we compared the output \( \tilde{P}(\phi) \) produced by cuFFS with a CPU implementation of RM synthesis from Brentjens (2007). The CPU and the GPU codes were used to process a data cube from the Westerbork Synthesis Radio Telescope Spitzer Infrared Nearby Galaxies Survey (WSRT-SINGS; Heald et al. 2009). We compared the output data cubes \( \tilde{P}(\phi) \) on a pixel-by-pixel basis and found that the mean of the ratio between the outputs produced by the GPU and the CPU to be 1.0 with a standard deviation of \( 1.5 \times 10^{-6} \) implying that the output produced by cuFFS matches the output produced by the CPU code very well.

Figure 5.6 shows the ratio of the pixels values produced by the GPU and the CPU code \( \frac{\tilde{P}_{\text{GPU}}(\phi)}{\tilde{P}_{\text{CPU}}(\phi)} \) plotted as a function of the output pixel values in \( \tilde{P}_{\text{GPU}}(\phi) \). As can be seen from Figure 5.6, the pixel values in the two output cubes match very well.

To judge the performance of cuFFS, we compared the execution time of our GPU code with a CPU implementation of RM Synthesis from Brentjens (2007)
Figure 5.6 – Top: Comparison between the output pixel values produced by cuFFS and a CPU implementation by Brentjens (2007). **Bottom:** Zoomed-in version of the figure shown in the left panel to better display the scatter in pixel values computed by the GPU and the CPU codes. The scatter seen for the low output pixel values is probably due to numerical error.
using a combination of both real and mock datasets from current and upcoming telescope facilities. A brief description of each test dataset is presented below:

- **WSRT**: As a typical Westerbork polarimetry dataset, we used one of the datacubes produced as part of the Westerbork SINGS survey (WSRT-SINGS; Heald et al. 2009). After removing frequency channels corrupted by RFI, the final datacube consists of 803 frequency channels. The frequency setup used in the WSRT-SINGS survey results in a Faraday depth resolution of $144 \text{ rad m}^{-2}$. Based on these values, the input and the output datasets have dimensions $512 \times 512 \times 803$ and $512 \times 512 \times 401$ respectively.

- **LOFAR\_LR**: Diffuse polarized emission from the Galactic foreground can be detected at 150 MHz in low resolution LOFAR maps with a resolution of a few arcmins (Jelić et al. 2015; Van Eck et al. 2017). The LOFAR low resolution dataset used for this test corresponds to a field around the nearby dwarf galaxy NGC 1569. The input datacubes are at a resolution of $250''$. Calibration and imaging procedure used to obtained the datacubes are discussed elsewhere in this thesis. After visual inspection and rejecting channel maps corrupted by RFI, the final datacube contains 856 frequency channels. With 48.8 kHz-wide channel maps covering the bandwidth ranging from 120.238 – 166.087 MHz, the output Faraday cubes will have a Faraday depth resolution of $1.17 \text{ rad m}^{-2}$ and sensitive to a maximum Faraday depth value of about $350 \text{ rad m}^{-2}$. Thus the dimensions of the input and the output datacubes are $540 \times 540 \times 856$ and $540 \times 540 \times 7000$.

- **LOFAR\_HR**: The LOFAR high resolution dataset was produced from the same visibility data that was used to create the LOFAR\_LR datacubes. The only difference between the LOFAR\_HR and LOFAR\_LR datacubes is that LOFAR\_HR are at a higher angular resolution of $26''$. The dimensions of the input and the output datacubes are $4000 \times 4000 \times 856$ and $4000 \times 4000 \times 7000$.

All CPU and GPU benchmarks were carried out on CSIRO’s Bracewell computer cluster. Each node on the cluster is powered by 28 Intel(R) Xeon(R) CPU E5-2690 v4 2.60GHz processors with 256 GB RAM and is equipped with four NVidia Tesla P100-SXM2-16GB GPUs. The CPU code was permitted to use upto 245 GB of RAM. The execution times for different datacubes using the CPU and the GPU implementation is listed in Table 5.1. Note that the execution times for the CPU code will increases linearly with decreasing RAM size.

From the execution times reported in Table 5.1, it can be seen quite clearly that our GPU implementation in both FITS and HDF5 mode is substantially faster than the execution times achieved using the CPU implementation. Execution times reported in Table 5.1 while using cuFFS in FITS mode indicate that rotating and derotating the input and output datacubes still dominates the total execution time. If, however, the data rotation is done as part of the imaging pipeline that produces the Stokes Q and U cubes, then the GPU implementation is about two orders of magnitude faster than the CPU implementation for large datasets.
Table 5.1 – Comparison of CPU and GPU execution times for different input and output data cubes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Spatial pixels</th>
<th>Channels</th>
<th>$\phi$ planes</th>
<th>Execution time [hh:mm:ss]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CPU (a)</td>
</tr>
<tr>
<td>WSRT</td>
<td>$512 \times 512$</td>
<td>803</td>
<td>401</td>
<td>00:02:02</td>
</tr>
<tr>
<td>LOFAR_LR</td>
<td>$540 \times 540$</td>
<td>856</td>
<td>7000</td>
<td>00:39:30</td>
</tr>
<tr>
<td>LOFAR_HR</td>
<td>$4000 \times 4000$</td>
<td>856</td>
<td>7000</td>
<td>58:01:32</td>
</tr>
</tbody>
</table>

Notes. CPU tests (single threaded, vectorized) were carried out on a single node with Intel(R) Xeon(R) CPU E5-2690 processor at 2.6 GHz clock with 256 GB RAM. The CPU code was permitted to use a maximum of 200 GB RAM. GPU tests was carried out using an NVIDIA Tesla P100-SXM2-16GB device. (a) FITS file format; (b) HDF5 file format; (c) Time spent reordering the input datacube to have the frequency axis as the first axis and reordering the output datacubes so that the Faraday depth axis is the third axis; (d) Compute time.
5.4 Conclusion and future outlook

In this chapter, we have presented a GPU-accelerated CUDA program to perform a widely used radio astronomy algorithm called rotation measure synthesis. The software package is made publically available through Github. The software package is capable of handling input and output datacubes in either the FITS or the HDFITS format. If needed, additional data formats can also be supported with very little change to the existing code base. In addition to rotation measure synthesis, we plan to implement advanced polarization processing tools like RM-CLEAN (Heald 2009) in the future version of our software package.

Comparing our GPU implementation on NVIDIA Tesla P100-SXM2-16GB GPU with single-threaded, vectorized CPU implementation, depending on the input and output format, we observe gains in execution times by at least a factor of 15 for smaller dataset from telescopes like WSRT and by at least by two orders of magnitude for larger datasets from telescopes like LOFAR and SKA1. Further speedup can be achieved by using Fast Fourier Transforms (FFT) instead of Discrete Fourier Transforms (DFT) in equations 5.13 and 5.14. CUDA accelerated Fast Fourier Transform can be accelerated using standard libraries like cuFFT\(^{14}\). We aim to implement these advanced features in future versions of cuFFS.

Finally, as discussed in section 5.3.2 and Table 5.1, the main bottleneck in our implementation of RM synthesis is the FITS file access using CFITSIO. Even for a small WSRT dataset, rotating the cubes to speedup I/O can be an order of magnitude longer than the actual compute time. In light of the large data cubes that will be produced by the current and upcoming radio telescopes, this I/O overload is certainly a cause for concern. To minimise the amount of time spent rearranging data, we make the following two recommendations to teams carrying out large polarization surveys:

1. Adopt data formats like HDFITS which benefit from combining the FITS data model with the HDF5 file format. Accessing an array stored in HDFITS format along the slowest-varying dimension can be two orders of magnitude faster than FITS format.

2. If surveys prefer to use the standard FITS format to store their data products, they should at least consider archiving the data products not intended for visualisation (like Q and U cubes) with the frequency axis as the first/fastest-varying axis.

\(^{14}\)https://developer.nvidia.com/cufft