Chapter 5
Computational Issues

ABSTRACT
LOFAR (LOw Frequency ARray) is a new and innovative effort to build a radio-telescope operating at the multi-meter wavelength spectral window. The electric signals from the LOFAR antennas are digitized, transported to a central digital processor, and combined in software in order to map the sky. One of the most exciting applications of LOFAR will be the search for redshifted 21-cm line emission from the Epoch of Reionization (EoR). It is currently believed that the Dark Ages, the period after recombination when the Universe turned neutral, lasted until around the Universe was 400,000 years old. During the EoR, objects started to form in the early universe and they were energetic enough to ionize neutral hydrogen. The precision and accuracy required to achieve this scientific goal, can be essentially translated into accumulating large amounts of data. In this Chapter we review the computational challenges and describe some aspects of the pipeline.

5.1 Data size
One of the most challenging aspects of the LOFAR EoR experiment is the large dynamic range between the different components of the sky signal. Discrete sources can be of the order of $10^{1-5}$ Jy/beam while the Galactic diffuse emission as well as the confusion amount to 5mJy/beam and 3mJy/beam respectively\(^1\). The noise in the data is of the order 10$\mu$Jy/beam, while the desired cosmic signal is of the order of 1$\mu$Jy/beam, where we assumed a synthesized beam resolution of 3 arcminutes at 150 MHz. Even after

\(^1\)We have assumed a PSF of three arcminutes, so that $1\mu$Jy/beam corresponds to 2 Kelvin.
very accurate foreground removal the EoR signal is still buried deep in the noise. To reach statistically detectable EoR signals, a long observation run of at least 400 hours is required.

The LOFAR EoR Key Science Project plans to observe up to five independent windows in the sky in order to support the statistical detection of the cosmic signal. For each window we plan to use three independent station beams and cover a bandwidth of 64 MHz with a resolution of 10 kHz. This yields 6400 channels. At each time step of 10 seconds, \( \sim 1200 \) full-polarization visibilities will be recorded. The total number of time-steps will be 3000-4000 per day. This will result in a recorded visibility dataset of the order of one to two petabytes, including calibration and flagging meta-data.

After the standard calibration (see introduction) the observed visibilities and ME parameters will be used for the inversion step described in Chapter 4. The numerical complexity of the algorithm is \( O(N^3) \), similar to other statistically optimal algorithms (like the maximum a-posteriori and the asymptotic likelihood methods). However, the latter algorithms are more efficient in parallelizing the data processing as they can treat snapshots of data independently and then combine the results. The two main parallelization axes are the frequency axis and the observational window axis. Parallelization over those two is trivial. This means that one has to deal with \( 10^9 \) visibilities per channel, with 192,000 channels for 5 windows and 6 beams.

### 5.1.1 Background

This immense amount of data is affected by instrumental corruptions, which will be determined, to first order, during the initial processing. This involves finding a good initial solution of the parameters for all instrument and sky effects using a modified SELFCAL loop and a simple model for for the sky (e.g. bright calibrator sources). Solving for the parameters is a highly non-linear process, bound to converge to secondary minima, if not carried out carefully.

The data model can be written as a set of linear equations \( \mathbf{v} = \mathbf{A}(\mathbf{p}) \mathbf{s} + \mathbf{n} \), where \( \mathbf{v} \) is the observed data vector, \( \mathbf{A}(\mathbf{p}) \) is a sparse matrix describing the instrumental effects, \( \mathbf{s} \) is the true underlying sky signal and \( \mathbf{n} \) is a vector representing uncorrelated, spatially white noise. The Maximum Likelihood solution to this problem is (Chapter 4):

\[
\mathbf{s}_{\text{ML}} = \left[ \mathbf{A}^\dagger(\mathbf{p}) \mathbf{C}_{\text{noise}}^{-1} \mathbf{A}(\mathbf{p}) \right]^{-1} \left[ \mathbf{A}^\dagger(\mathbf{p}) \mathbf{C}_{\text{noise}}^{-1} \right] \mathbf{v}.
\]

\( \mathbf{C}_{\text{noise}}^{-1} \) stands for the inverse covariance matrix of the noise. Solving this equation is essentially a linear algebra problem, but the solution is non-trivial because \( \mathbf{v} \) is a vector of \( 10^9 \) double-precision, complex numbers.

### 5.2 Regularization

The system of linear equations that we described in the previous section cannot be inverted directly. The resulting system matrix is singular for most practical cases and thus regularization has to be used in order to get an approximate solution that is close to the real one. In Chapter 4 we discussed two regularization methods: Tikhonov and diffusion.
operators. Due to the immense data size we are obliged to choose an implementation of each method that requires the smallest number of transfers from the disk to memory and from the host memory to the GPU memory. We will describe those two methods briefly:

### 5.2.1 Tikhonov

Using a Tikhonov functional the solution becomes:

\[
A^T C_{\text{noise}}^{-1} v = \left( A^T C_{\text{noise}}^{-1} A + \alpha I \right) s
\]

The matrix can be rewritten in block matrix notation as:

\[
\begin{pmatrix}
A^T (N^T)^{-1} & \sqrt{\alpha} I \\
\end{pmatrix}
\begin{pmatrix}
A N^{-1} \\
\sqrt{\alpha} I
\end{pmatrix}
\delta K =
\begin{pmatrix}
A^T (N^T)^{-1} & \sqrt{\alpha} I \\
\end{pmatrix}
\begin{pmatrix}
v \\
0
\end{pmatrix}
\]

or equivalently:

\[
B^T B \delta K = B^T \begin{pmatrix} v \\ 0 \end{pmatrix}
\]

The matrix \( B \) has the structure:

\[
B = \begin{pmatrix}
n_{11} a_{11} & n_{11} a_{12} & \cdots & n_{11} a_{1n} \\
\vdots & \vdots & \vdots & \vdots \\
n_{mm} a_{m1} & n_{mm} a_{m2} & \cdots & n_{mm} a_{mn} \\
\sqrt{\alpha} & \sqrt{\alpha} & \cdots & \sqrt{\alpha}
\end{pmatrix}
\]

\( B \) is neither completely dense nor sparse. The upper part is dense but the lower part is sparse. We use a column oriented transformation (Householder transformation, Golub \& Loan 1996) to estimate a transformation of the matrix \( B \) to an upper triangular matrix. The Gram-Schmidt methods is related to the Householder methods but it is less stable numerically. The first step of the factorization involves the annihilation of the non-zero elements of the first column below the main diagonal. All elements of this sub-matrix are affected at every iteration and the process continues until all elements below the main diagonal have been annihilated. Using the Householder reflections we can factorize the matrix in an \( R \) matrix which has the same size as \( A \) and into a \( Q \) matrix which is square with as many rows as \( A \). Then:

\[
(H_1, ..., H_n) v \equiv Qv = Rs
\]

The above algorithm is conceptually simple and dominated by matrix-vector multiplies. However, the number of computations per memory element fetched from global memory is quite low. To further improve performance, an algorithm in which several Householder transforms may be applied in a single operation was sought (Kerr et al. 2009).

The effect of the Tikhonov method on the eigenvalues can be seen in Figure 5.1. Regularization suppresses the lowest eigenvalues and thus the new system has a reduced
condition number. The correct choice of the regularization parameter is crucial as it can affect the cosmic signal in a non-trivial way. In Figure 5.2 we show the eigenspectrum of a simulated LOFAR map at a frequency of 150 MHz. The two red circles mark the inflection points. The first inflection point corresponds to the transition from the astrophysical signal subspace to a subspace representing both the EoR signal and the noise. The second inflection point corresponds to the true noise subspace. Figure 5.3 shows the reconstruction of the map using a different number of eigenmodes. By filtering the large singular values that correspond mostly to the foreground emission we can filter the foregrounds. The rest of the eigenmodes correspond to the EoR signal and effective noise. In Figure 5.4 we compare the recovery of the rms of the cosmic signal as a function of frequency using the SVD and the polynomial fit method of Jelić et al. (2008). The SVD method gives the correct level of the rms but has a higher error. This is because the EoR signal subspace cannot be completely separated from the subspace of the noise.

Figure 5.1: (top) The eigenvalues of the $A^T A$ for a simulated LOFAR EoR observation. (bottom) the eigenvalues of above matrix plus the regularization matrix.

Figure 5.2: The singular value spectrum of a simulated LOFAR map.
5.2 Regularization

Figure 5.3: Simulated LOFAR map reconstruction using 2, 5, 10 and 16 modes

Figure 5.4: Comparison of the polynomial extraction method of Jelić et al. (2008) with the SVD method.

5.2.2 Iterative regularization for lagged diffusivity

In recent years a new class of partial differential equation-based techniques has emerged in image restoration problems (Vogel & Oman, 1996). Rudin et al. (1992) introduced the popular technique of Total Variation (TV). TV can be posed as a variational problem, resulting to a highly non-linear Euler-Lagrange equation. However, this method is quite unstable and convergence is slow. Vogel & Oman (1996) proposed a linearization tech-
nique for this problem, which essentially resolves to the solution of a linear equation at each step. The algorithm is very robust and linearly convergent. Radio astronomical imaging using ML techniques can greatly benefit from such a method because it preserves sharp features, like point sources superimposed on a diffuse background, much better. An outline of the algorithm, adapted for the case of synthesis imaging, is presented here:

\begin{verbatim}
   i:=0
   s_0:=initial guess;
   begin fixed point iterations
   \hat{L}_i := \hat{L}(s_i); discretized diffusion operator
   g_i := A^T(A s_i - V_0 bs + \alpha \hat{L}_i s_i); gradient
   H := A^t A + \alpha \hat{L}_i; approximate Hessian
   d_{i+1} := -H^{-1} g_i; quasi-Newton step
   s_{i+1} = s_i + d_{i+1}; update solution
   increment iteration counter
\end{verbatim}

At each iteration a linear diffusion equation is solved to obtain the new iterate, based on the result of the previous step. Notice that a global line-search (like a line-search in many optimizations) is not needed to ensure convergence. Furthermore, from our numerical experiments we have observed that the methods needs a very small number of iterations (∼3) and adapts very well to data-parallel implementations, such as those relevant for GPUs.

### 5.3 Computational burden

The currently, less complex, processing-pipeline requires access to a 1000 CPU cluster for more than a 1 year, and analyzing the results requires a further 10-100 Tflop/s processing power for 1 year to perform ML inversions. However, the linear equations describing the LOFAR data-model lend themselves perfectly to be solved, not on classical CPUs, but on Graphical Processor Units (GPUs). We have implemented basic simulation, inversion and analysis codes on a mini-cluster of 3 NVIDIA-Tesla S870 units and in several tests we obtain GPU/CPU speed up ratios of 30-85 in the relevant linear operations, including I/O, similar to test by other groups.

More specifically, forming $A^T C_{noise}^{-1} A$ requires $10^{15}$ complex multiply-add (CMAD) operations. $A^T C_{noise}^{-1} v$ requires $\sim 2 \times 10^{10}$ CMADs and the solution of the ML equation requires $\sim 5 \times 10^{14}$ CMADs. Each CMAD requires 4 flops on a GPU. This amounts to $\sim 60$ Pflop per channel and $\sim 1.2$ Zetaflop for the whole data set. Our numerical tests have indicated that the total processing time per channel is 7 seconds and if we extrapolate to the total number of channels for the LOFAR EoR KSP, the relevant time is $\sim 100$ days.
Commodity GPUs are inexpensive resources for delivering very high computing throughput for certain classes of applications. GPUs are sold primarily as an integrated component in display adapters for desktop personal computers. High-throughput GPUs are primarily aimed for the video game market, but in the last couple of years there is an increased interest in using them for numerical computations. This fact has allowed GPU vendors to exploit micro-architecture parallelism for increased performance without constraint by the application and without requiring much architectural infrastructure to facilitate parallel execution. Simultaneously, GPU execution models have grown fast, in response to the needs of graphics programmers, thereby enabling a wide range of computing tasks. GPU vendors have consequently developed graphics-agnostic programming models such as NVIDIA’s Compute Unified Device Architecture (CUDA) and Open Compute Layer (OpenCL) to facilitate general purpose computing on GPUs. Nevertheless, fully exploiting the peak performance capacity of GPUs has remained a challenge. Algorithms with very high arithmetic intensity, very little need to synchronize between execution paths, and very few scatter operations (collect data from many addresses in memory to process in a single procedure call) typically perform well on GPUs without the need for careful optimization, but many computing tasks do not follow these idealized constraints. The final data processing of the EoR KSP resolves to a numerical linear algebra problem and can thus take advantage of such hardware accelerators.

To solve the equations that results from the ML inversion, described in Chapter 4, we use the QR decomposition method. Several algorithms for fast QR decomposition exhibit a high degree of parallelism, but have low arithmetic intensity and are highly coupled between execution paths, requiring synchronization between elements after small numbers of arithmetic operations. As a result, attempts to exploit GPUs to accelerate QR decomposition have only been moderately successful achieving 4-5x speedup.

We have carried a series of benchmarks using NVIDIA Tesla GPUs and the various LAPACK/BLAS implementation for GPUs.

Figure 5.5: The CUDA execution model: The host computer invokes a kernel than runs on the GPU. The kernel is executed as a collection of threads running on the GPU. The threads are organized in grids of thread blocks.
5.4.1 Benchmarks

The test systems is equipped with an Intel Q6600 Core 2 Quad processor with 4 MB of cache per core. The physical RAM amounts to 8 GB with a bus speed of 667 MHz (DDR2). The disk is a Western Digital WDC WD1600AAJS-0 with 8 MB of cache. We measured the read speed of the disk to be 118 MByte/s at a radial distance of 10% of the maximum radius of the rotating rigid platter of the disk. The IO performance variation with radius is typical for a SATA disk, but we did not go further away than a tenth of a radius in order to have consistent measurements. The S1070 was connected via a PCI Express x16 Gen. 2.0 bus. The OS was Centos Linux 5.3 and the kernel version was 2.6.18-128.1.10.el5 without any patches other than those of the maintainer. We used CUDA v2.3 and the 190.18 driver. We measured the values of the Host-to-Device, Device-to-Host and Device-to-Device memory transfer bandwidth and we found them to be 1322 MByte/s, 1295 MByte/s and 74245 MByte/s consistently.

5.4.2 Test runs

We identify two key operations that we need to perform:

- Sparse vector-matrix multiplications
- Matrix decompositions (and eigenvalue estimation)

For the first type of operations we used a slight modification of the code by N. Bell and M. Garlard described in the NVIDIA Technical Report NVR-2008-004 [Bell & Garland, 2008]. We used an input matrix of $1,748,122 \times 62,729$ double-precision elements with 6,804,304 nonzero values. We measured a performance of 6.7 Gflop/s and 14.4 Gflop/s with and without the usage of texture memory respectively. The matrix we chose was a worst case scenario as we expect even sparser matrices from LOFAR data.

For the matrix decompositions we used several codes. More specifically we used the code by Garland, the MAGMA\textsuperscript{2} project code, CULA\textsuperscript{3} tools and a custom written code performing LU factorization using the CUBLAS library.

At the moment the CULA library supports only single precision (unlike MAGMA) and a limited number of LAPACK of functions. We noticed that the U and V matrices in the SVD are different from those in Matlab. We subsequently used the LAPACK code by M. Garland. Finally, we used the MAGMA library. We were especially interested in the implementations of the block algorithm for Cholesky decomposition of positive definite matrices (*POTRF, BLAS lvl. 3). For a matrix of $1600^2$ elements we get a performance of 251 and 64 Gflop/s for single and double precisions floats respectively. The normalized errors are $1.27 \times 10^{-16}$ and $9.43 \times 10^{-8}$. The speedup versus the quad-core Q6600 CPU was 4.1 and 2.0 respectively. We also tested a matrix of $32000^2$ elements in single precision so that the full amount of the device RAM is allocated. The relevant numbers are 234.6 Gflop/s and $1.32 \times 10^{-7}$. The average efficiency for single and double precision compared to the peak performance of the GPU is 35 and 20 per cent respectively.

We also implemented our own version of LU factorization using the CUDA BLAS library

\begin{footnotesize}
\begin{itemize}
\item 2\textsuperscript{http://icl.cs.utk.edu/magma}
\item 3\textsuperscript{http://www.culatools.com}
\end{itemize}
\end{footnotesize}
in order to get hands-on experience on the programming environment. We experienced some issues with name mangling, but we managed to overcome those using the NVIDIA forums. We looked into the following issues as well, but without solid conclusions, given the limited testing time:

- **Zero-copy**: We actually transfer the data once from the disk to the memory and then perform the aforementioned operations. Zero-copy might prove useful especially if we manage to hide the latency.
• Asynchronous memory transfer

• Use of the multi-processor cache

• OpenMP: Using different GPUs to simultaneously process the real and imaginary part or process different chunks of data concurrently. We would like to look into a hybrid OpenMP/MPI solution in the future where the data are processed on several accelerators within a cluster.

• Portland Group compilers: S. Yatawatta tested the compiler for porting naive C code to run optimized on a GPU. Seems to work somewhat better, but the compiler is work in progress. Later versions should provide double precision, function calls, asynchronous transfer etc.

5.4.3 Conclusions

The results we presented above are just initial estimates. Careful fine-tuning of the kernels has to be done in order to tailor them for our specific needs. More specifically we still need to exploit ways to use the shared memory of the devices in a coalescent way, as well as tune the number of threads and blocks. Due to the large amount of transfers from the disk to memory and the memory to the device careful syncing of the threads after async. transfers from the disk is crucial. Nonetheless, we are quite satisfied with the performance and stability of the test system, reaching 20 per cent of the peak performance per node.