Random Fields Generation on the GPU with the Spectral Turning Bands Method

Lars Hunger[1,4], Biagio Cosenza[2], Stefan Kimeswenger[1,3], and Thomas Fahringer[2]

[1] Institute for Astro- and Particle Physics, University of Innsbruck, Austria
[2] Institute of Computer Science, University of Innsbruck, Austria
[3] Instituto de Astronomía, Universidad Católica del Norte Antofagasta, Chile
[4] BrainLinks-BrainTools, University of Freiburg, Germany

Abstract. Random field (RF) generation algorithms are of paramount importance for many scientific domains, such as astrophysics, geostatistics, computer graphics and many others. Some examples are the generation of initial conditions for cosmological simulations or hydrodynamical turbulence driving. In the latter a new random field is needed every timestep. Current approaches commonly make use of 3D FFT (Fast Fourier Transform) and require the whole generated field to be stored in memory. Moreover, they are limited to regular rectilinear meshes and need an extra processing step to support non-regular meshes.
In this paper, we introduce TBARF (Turning BAnd Random Fields), a RF generation algorithm based on the turning band method that is optimized for massively parallel hardware such as GPUs. Our algorithm replaces the 3D FFT with a lower order, one-dimensional FFT followed by a projection step, and is further optimized with loop unrolling and blocking. We show that TBARF can easily generate RF on non-regular (non uniform) meshes and can afford mesh sizes bigger than the available GPU memory by using a streaming, out-of-core approach. TBARF is 2 to 5 times faster than the traditional methods when generating RFs with more than 16M cells. It can also generate RF on non-regular meshes, and has been successfully applied to two real case scenarios: planetary nebulae and cosmological simulations.

Keywords: gpu, random field, turning band, fft, astrophysics, non uniform mesh, non-regular mesh, gpgpu, spectral methods

1 Introduction

A Random Field (RF) is a spatial distribution of correlated random values. One RF point consists of a random value, and its corresponding spatial coordinates. The correlation function describes how the values of RF points behave depending on their relative position to each other. For instance, for a correlation function with high correlation on short ranges, closeby points have very similar values. This leads to the formation of clusters of points with similar values. The size distribution of these clusters is described by the power spectrum. The correlation function and the power spectrum are two different ways to describe a RF. The

power spectrum can be transformed into a corresponding correlation function and vice versa, according to requirement of the Wiener-Khinchin theorem [27].

RF generation algorithms are of crucial importance for many scientific areas. They are used to generate initial conditions for cosmological structure formation simulations like the Millenium simulation [5], to create winds in planetary nebulae simulations (see Sec.6) and for the initialization of N-body simulations [19]. In simulations that use a turbulence driving technique like the one proposed in [8], a RF has to be generated in each time-step of the Magneto-hydrodynamical simulation. RFs are also often used in geostatistical research [24] together with a technique called Kriging for creating topological maps. In other words, RFs are used when the statistical properties of a scalar field are known and distinct realizations are to be generated.

We focus on three-dimensional (3D) RF. Traditional approaches to compute 3D RFs make extensive use of 3D Fast Fourier transforms (3D FFT). These 3D FFT-based methods are limited to regular meshes for generating random fields.

In this paper we introduce TBARF (Turning BAnd Random Fields), a new random field generation implementation based on the Turning band (TB) method that has been highly optimized to run on GPUs. The proposed algorithm replaces the 3D FFT used in a traditional approaches with a two step approach: a faster, lower dimensional FFT to generate lines (which uses a smaller set of points with respect to the traditional approach); and a multi-dimensional projection step, where all of the lines affect each mesh point of the random field. TB RF generators are not commonly used for generating large RFs since, on the CPU, they are much slower than a traditional 3D FFT approach. TB methods are slower, on the CPU, since each grid point is affected by all off the lines, while in the 3DFFT approach the field is generated in one pass. In this work we demonstrate that TB methods can be highly optimized for GPUs and allow the out-of-core generation of RF on regular and non-regular meshes.

The contribution of this paper are as follows:

- TBARF, a TB-based RF generation algorithm optimized for GPUs exploiting loop blocking and unrolling;
- Support for the fast generation of RF on irregular meshes;
- Out-of-core streaming computation of a RF which allows the generation of a very large RF, not possible with the traditional approach on the GPU;
- Practical application of TBARF to two real test cases: planetary nebulae and cosmological simulations.

2 Related work

Random field generation The TB method itself was first proposed in [23]. The spectral TB method was then first proposed in Mantoglou [3] where a TB method like TBARF is first described in combination with a spectral line generation algorithm. A Matlab version of the TB method can be found in Xavier et al. [4].

GPU Graphics Processing Units (GPUs) are used not only for 3D graphics rendering but also in general-purpose computing because of their huge computa-

tional power. GPUs' programmability has significantly improved thanks to high-level parallel programming languages such as the CUDA [1] and OpenCL [2]. The GPUs' huge potential computational power comes with some drawbacks: The available device memory is limited to few GBs (e.g. 6GB on NVIDIA Tesla K20); it requires slow host-device communications for big datasets. Moreover, optimizing code for GPUs means writing algorithms which are better suited for the hardware, but also exploring low level optimizations. Traditional compiler optimizations such as loop tiling (blocking) [16] and loop unrolling [14] have been successfully tested on GPUs [17, 15]. However, the search space is quite big [21, 18] and highly optimized codes still requires manual, problem-specific exploitation of the optimization space.

FFT Our work also focuses on one- and multi-dimensional FFT methods. For small-scale FFTs, if the data can be held entirely on a GPU, the computation can benefit from the high device memory bandwidth [12, 13, 10, 11]. However, if the data does not fit the available device memory, the overhead to transfer data between host memory (i.e. the CPU main memory) and device memory is a bottleneck [22]. This problem applies whenever the dataset is bigger than the available device memory, e.g. out-of-core computation or cluster computing [22].

3 The Turning Band Method

Correlation function and power spectrum The (auto-)correlation function describes the correlation of two values of a RF depending on their spatial positions. The power spectrum describes the size distribution of clusters in the RF. For well behaved correlation functions these two ways of describing a RF are interchangeable. This transformation is not always possible, but TBARF is able to create a RF from both a spectral density or a correlation function. The TB method is an asymptotically correct approach of generating multidimensional RFs which we use for generating 3D RFs. The TBARF algorithm has multiple steps. First, discrete 1D RFs, i.e. lines, have to be generated. The correlation function or the power spectrum that the 1D lines have to follow is calculated by

$$C_{1D}(r) = \frac{d}{dr}[r \cdot C_{3D}(r)]$$
$$S_{1D}(\omega) = \frac{4\pi \left|\omega^2\right|}{6} \cdot S_{3D}(\omega)$$

where C_{3D} is the correlation function, S_{3D} the power spectrum of the 3D field to be generated, r the distance between points and ω the angular frequency corresponding to a structure of a certain size. To generate these lines according to a power law power spectrum, we use a simple 1D Fourier transform approach [25]. For lines with an arbitrary power spectrum we use a pulse train method [26]. Lines according to a correlation function are generated using a circulant embedding approach [6].

Number of lines and line directions The TB method is an approximate method. The statistical quality depends on the number of lines used to create the multi-dimensional field. Empirical studies have shown that for a 3D field of any size

Algorithm 1 Turning bands method.

$S \leftarrow computeHaltonSequence()$
$Dir \leftarrow computeLineDirection(S)$
$L \leftarrow computeLines(Y)$ // requires 1D FFT
for all $line \in L$ **do**
    for all $cell(x, y, z) \in domain$ **do**
        $lineCoord \leftarrow -(x, y, z) \cdot Dir[line]$
        $linePoint \leftarrow round(lineCoord \times resolutionFactor) + lineLength \times 0.5 + 1$
        $index = line.index * linelength + linepoint$
        $value = L[index]$
        $field[index] = field[index] + value$
    end for
end for

1000 lines are sufficient to avert banding artifacts [3, 4]. A schematic picture of the TB method is shown in Fig. 1(right). The lines are laid out along unit vectors (u_i), starting at the origin, so that the surface of the unit sphere is covered as uniformly as possible. We create the unit vectors with the help of a pseudo-random Halton sequence, which leads to a closer to optimal coverage of the unit sphere than random vectors. After the direction vectors have been created, we rotate all vectors together by a random angle around the three major Cartesian axes. This assures that we do not produce statistical artifacts if we generate a large number of fields.

Projection step The last step is the projection in which the 3D RF is generated (Fig. 1(right)). A point P of the 3D RF is generated by projecting its location vector X_P onto the line i and adding the corresponding value of this line $L_i(P)$ to the value of the point P. For P, this projection is then repeated for each line. After doing the projection step for each point, we have generated the full 3D RF.

Traditional 3D FFT method As a comparison, we also show a traditional 3D Fourier Transform algorithm for creating a RF with a power law power spectrum and a power law index between -3 and -5. This algorithm is much less versatile than our TB algorithm. For the input data we choose the amplitude A for each 3D wavevector $\mathbf{k}$ according to the desired power spectrum. For each wavevector we also choose a random phase Φ to be able to generate different realizations of the RF. We choose the random phases of our input data so that $\Phi(\mathbf{k}) = -\Phi(\mathbf{-k})$, making sure that the result of the following inverse Fourier transformation is real. After filling the 3D array with the input data ($A \cdot \Phi$) we only have to perform a 3D inverse Fourier transformation on the array to get our final field with the correct power spectrum. With the inverse Fourier transform, contributions with different wavevectors are summed up according to their amplitude to generate a real valued field (see Fig. 1(left)). For the power law indexes outside the range -3 to -5, this method does not work because the resulting field will show very strong generation artifacts. There are more complex 3D FFT methods that can generate RF according to arbitrary power spectra but that is beyond the scope of this paper. To compare the results of both methods, we calculate the power spectrum

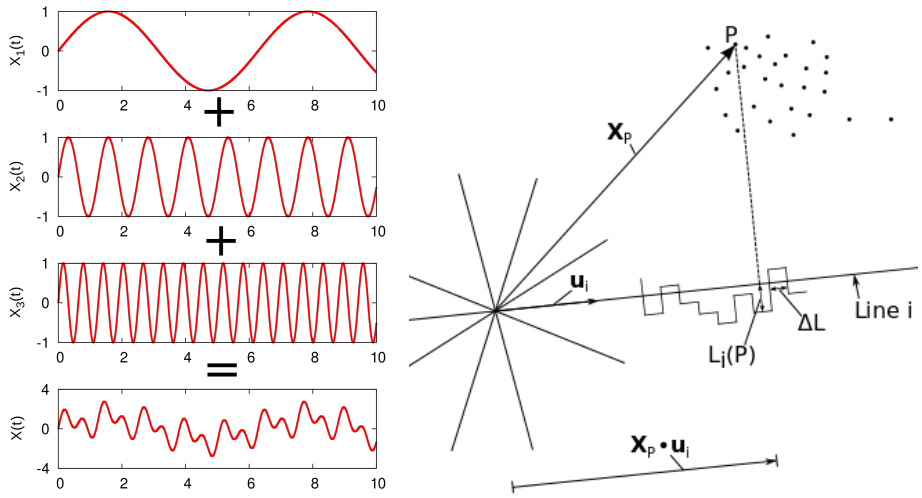

Fig. 1. In the FFT method (left), components with different frequencies (or wavevectors) are summed up according to their amplitude. This summing is done by performing the inverse FT. In The TB algorithm (right) the point positions P are projected onto the lines $X_p \cdot u_i$, and the corresponding values $L_i(P)$ are then summed over all lines.

of the resulting field and compare it with the theoretical power spectrum we aimed to generate. Both methods generate RFs with the correct power spectrum.

Non-regular (Non-uniform) Fields One advantage of the TB method is its ability to generate RF on non-regular meshes. The difference between regular and non-regular meshes is shown in Fig. 2. The 3D FFT methods can only generate RF on regular rectangular meshes since FFT works only on equally spaced points. In the projection step, the TB method can generate RF with arbitrary point positions. The resolution of the 1D lines has to be chosen high enough so that the smallest distance between two grid points can be sufficiently resolved. The ability to create RFs on non-regular meshes makes TBARF a very versatile RF generator. It can be used to create RF on regular grids with different resolutions like in Adaptive Mesh Refinement(AMR) or on entirely unstructured grids. Both of these tasks are much harder to perform with traditional 3D FFT methods.

4 Parallelization and Optimizations

The TB method, as described by Algorithm 1, comprises four main steps: the Halton sequence (line 1) and line direction generation (line 2), the one-dimensional field generation (line 3) , and the final projection step (lines 4-11). Step 1 and 2 are fast. Step 3 includes multiple 1D FFT calls with very small sizes, which are quite fast (cuFFT has an optimized $cufftPlanMany$ function for this). Therefore, the *projection code* is the main bottleneck and is where we focus our optimization efforts. In the following section we describe how we map that algorithm, and in particular the projection phase, onto the GPU hardware.

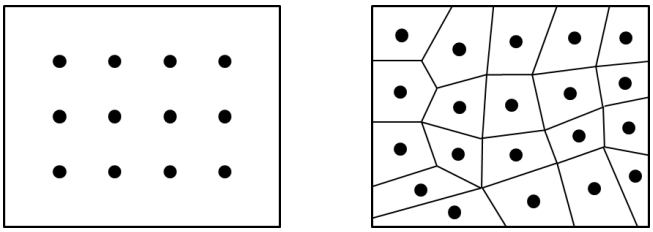

Fig. 2. Grid Points of a regular (left) and non-regular (right) mesh. In the irregular mesh the shape of the corresponding Voronoi cell is shown additionally.

OpenCL We use the OpenCL [2] model and terminology: the platform model comprises of a *host* connected to one or more *devices* (e.g. a GPU). Each device consists of one or more compute units (CUs) which are further divided into processing elements (PEs). A program running on a device is called *kernel*, and represents the parallel part of an OpenCL application. A single OpenCL thread is called *work-item*. Several work-items form a *work-group*. OpenCL provides a fast *local memory* which is shared between work-items belonging to the same work-group. Similarly, OpenCL offers fast local synchronization between work-items inside the same group. Host and device exchange data through memory buffers, which are passed as arguments to the kernel before its execution.

Parallelization strategy Algorithm 1 can be parallelized in two different ways. Following the original sequential formulation, it is possible to run a different OpenCL work-item for each line (*line parallelization*). Alternatively, it is possible to apply a loop interchange between the two for loops, therefore mapping a different OpenCL work-item to each cell, i.e. *cell parallelization*. The *line parallelization* approach has two drawbacks. First, writing cell values happens concurrently from different threads, therefore requiring an atomic addition. Unfortunately, atomic addition for double floating point precision is not included in OpenCL 1.1, but can be implemented by exploiting a 64-bit compare and exchange operation (*atom_cmpxchg*). However, atomic operations are extremely expensive on GPUs. The second drawback is the lower parallelism: while applying our approach to a real dataset, the number of lines is too low (ranging from 1024 up to 8192) to exploit GPUs' massively parallel architecture. On the other hand, cell parallelization exposes a high level of parallelism and does not require the use atomic operations. We tested the two parallelizations on a 128^3 mesh with 1024 lines of length 2600, where the cell parallelization was 50 times faster than the line parallelization.

```
1  __kernel void make_reg_field(int nr_lines,
2          int dim_x, int dim_y, int dim_z, int linelength,
3          __global double4* dir, __global double* L,
4          __global double* RF, double resfactor) {
5    const size_t dim_yz  = dim_y*dim_z;
6    int gid = get_global_id(0);
7    int k = gid / dim_yz;
8    int j = (gid % (fielddim_yz  )) / fielddim_y;
9    int i = gid - j * dim_y - k * dim_yz;
10   double4 id4 = {k, j, i, 0};
11   double rf_value = 0;
12   for(int l=0; l<nr_lines; l++) {
```

```
13      double linecoord = - dot(id4, dir[l]);
14      size_t linepoint = round(linecoord*resfactor)+linelength*0.5+1;
15      rf_value += L[l*linelength+linepoint];
16    }
17    RF[gid] = rf_value;
18  }
```

Listing 1.1. Non optimized OpenCL kernel for the cell parallelization projection kernel.

Loop blocking and unrolling Starting from the cell parallelization, we applied two loop optimizations to the for loop in line 12 (Listing 1.1). First, we tried to apply *loop blocking* (i.e. tiling) by partitioning the loop iteration space into smaller blocks (matching the work-group size), to ensure data used in a loop stays in the fast local memory available on the GPU. This technique can be applied to the line *dir* vector (line 13) which has coalesced memory accesses. However, the L array (line 15) is accessed randomly and cannot be prefetched.

We also applied *loop unrolling* (i.e. unwinding) to the same loop. The goal of loop unrolling is to reduce the number of iterations and branch penalties, as well as hiding memory access latencies while reading data from the memory [14]. The latter is particularly important in our case, as the inner loop performs many random accesses to the (slower) global memory. We applied to the projection code all the combinations of loop blocking and unrolling, with group size of 64, 128, 256 and 512, and unroll factors of 1, 2, 4 and 8.

Streaming out-of-core field generation GPU architecture has a limited amount of memory with respect to the RF size needed in some applications (already 30 GB for an 1024^3 grid). Especially while working with astrophysical datasets, RFs commonly exceed the memory available on a single GPU. This is a limitation for the standard approach based on 3D FFT [22]. Our approach only requires the lines to be stored on the GPU, and can be further distributed to work over multiple devices (e.g. on a multi-GPU or cluster of GPUs) or to perform an out-of-core streaming computation of the field in a single machine. TBARF splits the field in fragments of 128^3 cells to allow out-of-core RF generation.

Non-regular fields The TB method can also be used to generate a non-regular RF. We applied the same optimizations to a non-regular version of the projection kernel (note that other parts of the algorithm do not change), and tested different point distributions.

5 Results

Test settings We ran different versions of the TBARF code on a Intel Core i7 CPU 960 (3.20GHz 4 cores, 8 logical procs) and an NVIDIA GeForce GTX 550 (with 1280MB of OpenCL global memory). All tests were performed with double precision. OpenCL drivers were Intel OpenCL 1.2 SDK, OpenCL 1.1. CUDA and CUDA Driver API 5.5 (CC 2.0). We used the libWater CUDA extension [20] to support both CUDA and OpenCL kernels. For the FFT implementations, we used FFTW [9] on the CPU and CUFFT [11] for the CUDA version.

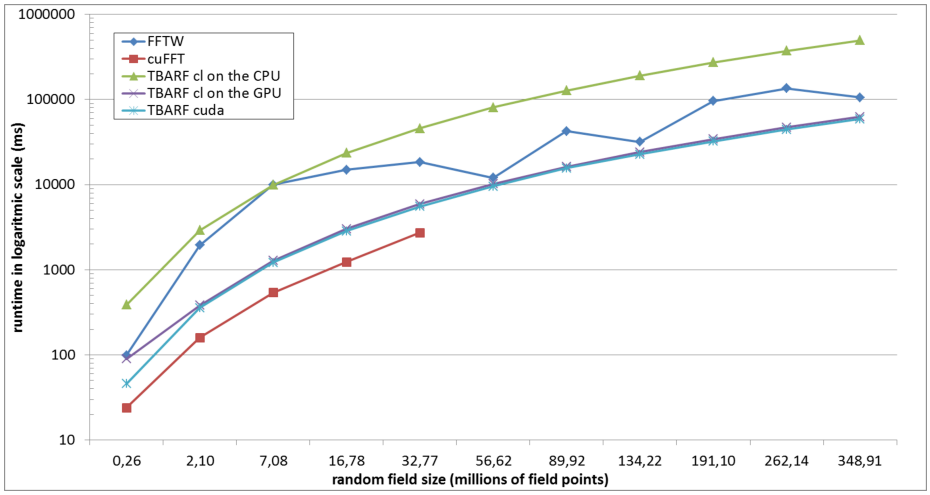

Fig. 3. Performance behavior of our out-of-core RF generation on different target architectures with varying problem sizes (i.e. the number of cells).

TBARF vs traditional approach We compared the traditional approach based on 3D FFT with our approach running on the GPU and CPU. Figure 3 shows the performance for different grid sizes and line lengths. For all the tests, we used 1024 lines and line length scaling according to the the grid size (e.g. 512^3 cells line length is 1064). The standard approach on the CPU uses 3D FFTW and supports very big grid sizes. The erratic behavior of the FFTW approach can be explained by the different algorithms employed by the FFTW library when the number of points is not equal to a power of two. The GPU version of the same approach based on cuFFT is faster, but it is limited by the amount of memory available on the GPU (up to 32.77 million cells for our test cases). 3D FFT methods require an extra cell per dimension (i.e. to generate a field of 256^3 elements we need a 257^3 3D FFT). We tested TBARF OpenCL on both CPU and GPU, and a CUDA version on the latter. Each TBARF code was running on its optimized configuration (see next paragraph). Despite being slower than the 3D cuFFT for small datasets, the TBARF CUDA version can quickly generate RFs bigger than the available device memory; on such datasets, it is always faster than the 3D FFTW approach. TBARF CUDA is about 4 to 6% faster than TBARF OpenCL on the NVIDIA GPU.

Projection kernel optimizations Table 1 shows the runtimes for the projection kernel on a uniform mesh generation with 128^3 cells. The use of local memory highly improves performance of GPU kernels, in the projection kernel this optimization can only be applied to the relatively small line buffer. Unfortunately there is no simple way to apply the same optimization to the line array. Applying both loop unrolling and blocking is not always beneficial for the CPU. On the GPU, the fastest CUDA configuration uses loop unrolling (factor 4) while the fastest OpenCL configurations utilize both loop unrolling and blocking.

Table 1. Runtime, averaged over multiple runs, of 32 different optimization configurations of the projection kernel. In bold, the best configurations for each target platform. Runtimes are in ms.

	non optimized				blocking			
local size	64	128	256	512	64	128	256	512
CL CPU	3600	3594	3590	3603	2966	**2945**	3038	3329
CL GPU	391	391	396	416	387	**386**	388	392
CUDA	369	368	369	368	**363**	365	366	369

	loop unrolling											
local size	64	128	256	512	64	128	256	512	64	128	256	512
unroll factor	2	2	2	2	4	4	4	4	8	8	8	8
CL CPU	3702	3628	3634	3632	3510	3516	3510	3468	3556	3510	3523	3515
CL GPU	388	387	389	387	**386**	388	388	393	387	**386**	389	394
CUDA	368	369	369	371	**363**	364	366	364	364	364	364	364

	loop unrolling and blocking											
local size	64	128	256	512	64	128	256	512	64	128	256	512
unroll factor	2	2	2	2	4	4	4	4	8	8	8	8
CL CPU	3248	3130	3121	3121	3098	3105	3064	3048	3113	3079	3051	3023
CL GPU	388	390	400	417	**386**	388	390	403	**386**	**386**	**386**	391
CUDA	369	372	385	412	367	369	372	381	364	367	366	370

Non-regular fields Finally, we tested the non-regular version of the RF generation algorithm against different mesh structures in order to understand how the point distribution affects the locality of the memory accesses. The first, named *regular*, has exactly the same distribution of the regular, uniform grid used before. The second uses a *jitter* sampling approach where each point has a regular position plus a random offset. The third is a completely *random* point distribution, where two close points in the input array may be very distant in space. Figure 4 shows that regular and jitter distribution are very similar in performance. However, the random distribution is noticeably slower than a regular one (10 to 25% slower) as it exposes poor memory accesses locality.

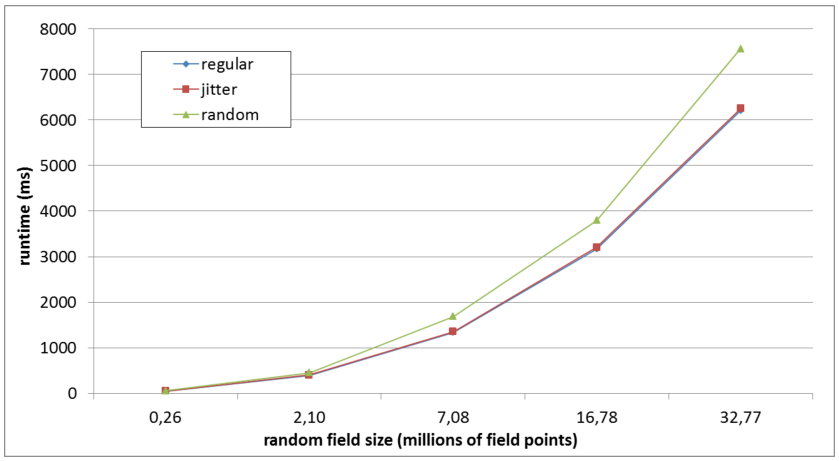

Fig. 4. Non-regular field with three different point distributions.

6 Applications

Astrophysics: Planetary Nebulae The code presented here has already been implemented to create a wind with density fluctuations in a Planetary Nebulae clump simulation. To have an inflowing wind entering on one side of the computation domain we create a RF tube of size 256 x 256 x 5000 with a power law power spectrum. The size of the tube will be larger for higher resolutions. For this problem we already use the out-of-core version of TBARF since the whole field is too large to fit into the main memory. Examples of the fields used can be found in Fig.5, for these simulations the power law index of the power spectrum is a free parameter, so we show RFs for different power law indices. With the optimized out-of-core CUDA kernel it takes 28241 ms to generate a RF with 256 x 256 x 5000 points using 1024 lines with a linelength of 4350 .

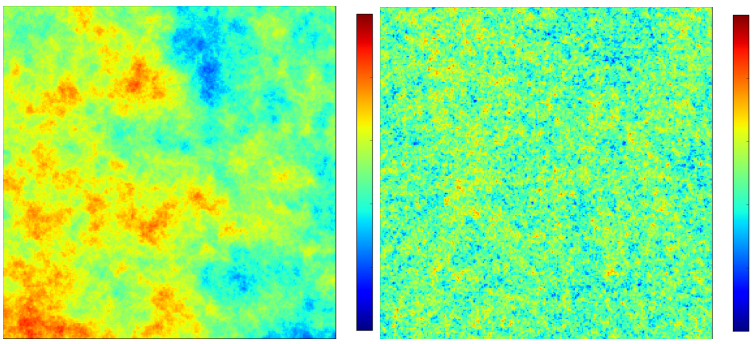

Fig. 5. 2D plane slices through 3D RF used in the Planetary Nebulae simulations. Red values are positive while blue values are negative. (left) shows a field with a power spectrum $P(k) \propto k^{-3.9}$ that emphasizes larger structures while (right) shows a field with a power spectrum of $P(k) \propto k^{-2.0}$ where larger structures are less prominent.

Astrophysics: Cosmology Simulations In the astrophysical community moving mesh techniques for calculating hydrodynamical simulations have become more popular. The most prominent example is AREPO, the new moving mesh n-body code by Volker Springel [7]. In these codes hydrodynamic simulations are performed on a non-regular mesh.TBARFs ability to create RFs on a non-regular mesh is a clear advantage over the traditional 3D FFT methods for all simulations performed with these moving mesh codes.

TBARF is able to generate RFs that can be used as initial conditions for cosmological structure formation simulations with AREPO. A realization of such a RF following a Harrison Zeldovich spectrum is shown in Fig. 6 (left). These new moving mesh codes can also be used to perform turbulence driven simulations. These simulations are typically quite large so the ability of TBARF to create the fields out-of-core is another advantage. A RF is needed in every time-step, making the RF generation a major contributor to the computational cost of the whole simulation. Until now the runtime of TB methods prohibited them from being used in this manner. With the increased performance on the GPU, TB

methods, like TBARF, are now a viable option for turbulence driven simulations on non-regular meshes. In Fig. 6 (right) we show a slice of a RF that can be used for this kind of turbulence driven simulations. With the optimized out-of-core CUDA kernel it takes 22803 ms to generate a RF with 512^3 points using 1024 lines with a linelength of 1065.

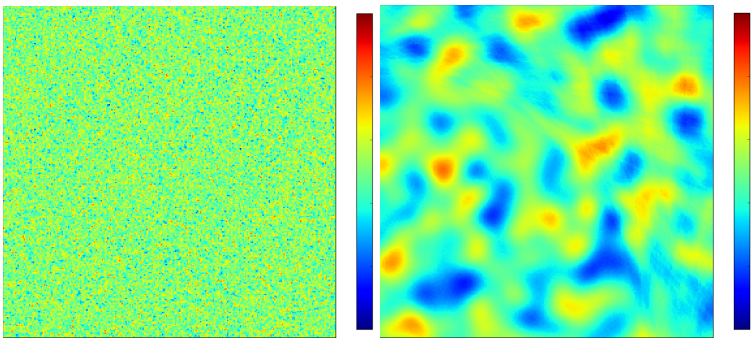

Fig. 6. Red values are positive while blue values are negative. (left) shows a 2D slice through a 3D RF with a power spectrum of $P(k) \propto k^{1.0}$ sometimes proposed as the initial fluctuations (Harrison Zeldovich Spectrum) of cosmological structure formations. (right) shows a slice through a 3D RF with a power spectrum of $P(k) \propto k^6 \cdot e^{(-k)}$ that is used for turbulence driving simulations.

7 Conclusions

In this paper we demonstrated that TB methods can be significantly sped up by porting them onto the GPU. We present TBARF, our implementation of the turning band method. TBARF efficiently generates random fields on both regular and non-regular meshes on the GPU. We showed that TBARF is able to create random fields which are bigger than the available device GPU memory quickly, thanks to its ability to do out-of-core streaming computation. Traditional methods based on 3D FFT are limited to the available device memory and can not generate random fields on non-regular meshes. These advantages make TBARF much better suited to be used in combination with, for example, moving mesh hydrodynamic codes than traditional 3D FFT RF generators. The project source is available at `https://github.com/LarsHunger/TBARF` under the LGPL License.

Acknowledgment This project was funded by the FWF Doctoral School CIM Computational Modelling under contract W 1227-N16 (DK-plus CIM) and by the Austrian Research Promotion Agency under contract 834307 (AutoCore).

References

1. NVIDIA : CUDA Compute Unified Device Architecture Reference Manual
2. Khronos OpenCL Working Group : The OpenCL Specification 1.1
3. A. Mantoglou : Digital Simulation of Multivariate Two- and Three-Dimensional Stochastic Processes with a Spectral Turning Bands Method. Mathematical Geology, Vol.19, No.2, 129–149 (1987)
4. X. Emery and C. Lantuéjoul : TBSIM: A computer program for conditional simulation of three-dimensional Gaussian random fields via the turning bands method Computers & Geosciences, Vol. 32, 1615-1628 (2006)
5. V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P.Thomas, H. Couchman, A. Evrard, J. Colberg and F. Pearce : Simulations of the formation, evolution and clustering of galaxies and quasars Nature 435, 629–636 (2005)
6. C. R. Dietrich and G. N. Newsam : Fast and Exact Simulation of Stationary Gaussian Processes through Circulant Embedding of the Covariance Matrix SIAM Journal on Scientific Computing, Volume 18 Issue 4, 1088-1107 (1997)
7. V. Springel : E pur si muove: Galilean-invariant cosmological hydrodynamical simulations on a moving mesh Monthly Notices of the Royal Astronomical Society, Volume 401, Issue 2, pp. 791-851 (2010)
8. J. Stone : Direct Numerical Simulations of Compressible Magnetohydrodynamical Turbulence Interstellar Turbulence, Proceedings of the 2nd Guillermo Haro Conference. Cambridge University Press, 1999., p.267
9. M. Frigo and S. G. Johnson : The Design and Implementation of FFTW3 Proceedings of the IEEE, Volume 93, Number 2, pp. 216–231 (2005)
10. V. Volkov and B. Kazian : Fitting FFT onto G80 Architecture Report, University of California, Berkeley (2008)
11. NVIDIA : CUDA CUFFT Library, Version 2.3, (2009)
12. N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. : High performance discrete fourier transforms on graphics processors. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC), pp. 2:1–2:12 (2008)
13. A. Nukada and S. Matsuoka : Auto-tuning 3-D FFT Library for Cuda GPUs. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC), pp. 30:1–30:10 (2009)
14. V. Sarkar : Optimized Unrolling of Nested Loops International Journal of Parallel Programming, Volume 2, Number 5, pp. 545-581 (2001)
15. Y. Yang, P. Xiang, J. Kong and H. Zhou : A GPGPU compiler for memory optimization and parallelism management Proceedings of the 2010 ACM SIGPLAN PLDI, pp. 86–97 (2010)
16. M. Wolfe : More Iteration Space Tiling Proceedings of the ACMIEEE Conference on Supercomputing, pp. 655–664 (1989)
17. G. S. Murthy, M. Ravishankar, M. M. Baskaran, P. Sadayappan : Optimal Loop Unrolling For GPGPU Programs IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–11 (2010)
18. K. Kofler, I. Grasso, B. Cosenza and T. Fahringer : An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, pp. 149–160 (2013)
19. K. Kofler, D. Steinhauser, B. Cosenza, I. Grasso, S. Schindler and T. Fahringer : Kd-tree Based N-Body Simulations with Volume-Mass Heuristic on the GPU Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)
20. I. Grasso, S. Pellegrini, B. Cosenza and T. Fahringer : LibWater: Heterogeneous Distributed Computing Made Easy Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, pp. 161–172 (2013)
21. H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch : A Multi-Objective Auto-Tuning Framework for Parallel Codes Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 10:1–10:12 (2012)
22. Y. Chen, X. Cui and H. Mei : Large-scale FFT on GPU Clusters Proceedings of the 24th ACM International Conference on Supercomputing (ICS), pp. 315–324 (2010)
23. G. Matheron: The intrinsic random functions and their application. Adv. Appl. Prob., 5:439-468 (1973)
24. J. P. Chiles and P. Delfiner : Geostatistics: Modeling Spatial Uncertainty. New York: John Wiley & Sons. (1999)
25. N. J. Kasdin and T. Walter: Discrete Simulation of Power Law noise. Frequency Control Symposium, 1992. 46th., Proceedings of the 1992 IEEE, 274-283 (1992)
26. M. Carrettoni and O. Cremonesi: Generation of noise time series with arbitrary power spectrum. Computer Physics Communications, Volume 181, Issue 12, p. 1982-1985. (2010)
27. S. Engelberg : Random signals and noise: a mathematical introduction. CRC Press. pp. 130 (2007)