GPU-accelerated Simulation of Massive Spatial Data based on the Modified Planar Rotator Model

Milan Žukič · Michal Borovský · Matúš Lach · Dionissios T. Hristopoulos

Received: date / Accepted: date

Abstract A novel Gibbs Markov random field for spatial data on Cartesian grids based on the modified planar rotator (MPR) model of statistical physics has been recently introduced for efficient and automatic interpolation of big data sets, such as satellite and radar images. The MPR model does not rely on Gaussian assumptions. Spatial correlations are captured via nearest-neighbor interactions between transformed variables. This allows vectorization of the model which, along with an efficient hybrid Monte Carlo algorithm, leads to fast run
times that scale approximately linearly with system size. In the present study we take advantage of the short-range nature of the interactions between the MPR variables to parallelize the algorithm on graphics processing units (GPU) in the CUDA programming environment. We show that the GPU implementation can lead to impressive computational speedups, up to almost 500 times on large grids, compared to the single-processor calculations. Consequently, massive data sets consisting of millions data points can be automatically processed in less than one second on an ordinary GPU.

**Keywords** Spatial interpolation · Hybrid Monte Carlo · Non-Gaussian model · Conditional simulation · GPU parallel computing · CUDA

1 Introduction

As massive remotely sensed spatio-temporal datasets are becoming more and more common, scalable statistical methods are needed for their efficient (preferably real time) processing. In particular, the data often include gaps that need to be filled to obtain continuous maps of observed variables so that environmental managers can make prompt and informed decisions and in order to avoid the adverse missing-data impact on statistical estimates of means and trends (Sickles and Shadwick, 2007). The gap-filling can be accomplished using various spatial interpolation techniques. However, most of traditional methods, such as kriging (Wackernagel, 2003), are computationally too expensive to handle such an ever-increasing amount of data. More recently several modifications have been implemented (Furrer et al., 2006; Cressie and Johannesson, 2018; Hartman and Hössjer, 2008; Kaufman et al., 2008; Ingram et al., 2008; Zhong et al., 2016; Marcotte and Allard, 2018), which increased the computational efficiency.
Some alternative approaches inspired from statistical mechanics, have been proposed to alleviate the computational burden of kriging methods. Models based on Boltzmann-Gibbs exponential joint densities that capture spatial correlations by means of short-range interactions instead of the experimental variogram (Hristopulos, 2003; Hristopulos and Elogne, 2007; Hristopulos, 2015), are computationally efficient and applicable to both gridded and scattered Gaussian data. To also overcome the restrictive joint Gaussian assumption the concept was further extended to non-Gaussian gridded data by means of non-parametric models based on classical spin models (Zuković and Hristopulos, 2009a,b) and generalizations that involve geometric constraints (Zuković and Hristopulos, 2013a,b).

Nevertheless, as long as the computations are performed in a serial manner, the above methods fail to fully exploit the available computational resources. With new developments in hardware architecture, another possibility to overcome the computational inefficiency is provided by parallel computation. Multi-core CPU and GPU hardware architectures have now become a standard even in common personal computers. Thus, the parallel implementations of various interpolation methods is a relatively cheap method to boost their efficiency. So far, several parallel implementations were applied to the most common interpolation methods, such as kriging and inverse distance weighting (IDW) on high performance and distributed architectures (Kerry and Hawick, 1998; Cheng et al., 2010; Guan et al., 2011; Pesquer et al., 2011; Hu and Shu, 2015) and general-purpose computing on graphics processing units (GPU) (Xia et al., 2011; Tahmasebi et al., 2012; Cheng, 2013; de Ravet et al., 2014; Mei, 2014; Stojanovic and Stojanovic, 2014; Mei et al., 2017; Zhang et al., 2018a,b).

These studies have demonstrated that significant computational speedups, up to nearly two orders of magnitude, can be achieved over traditional single CPU calculations by means of parallelization.
In the present paper we demonstrate the benefits of parallel implementation on GPU of the recently introduced gap filling method that is based on the modified planar rotator (MPR) model (Žuković and Hristopulos, 2017). The MPR model has been shown to be competitive with respect to several other interpolation methods in terms of the prediction performance. It is also promising for the automated processing of large data sets sampled on regular spatial grids, typical in remote sensing, due to its computational efficiency and ability to perform without user intervention. In the MPR model, parallelization is feasible due to the short-range (nearest-neighbor) interactions between the MPR variables (spins). Recent developments in spin model simulations (Weigel, 2011, 2012) have demonstrated that significant speedups can be achieved by using a highly parallel architecture of GPUs. In the present paper we show that the GPU implementation can lead to an enormous computational speedup, almost by 500 times compared to single-processor calculations, for massive data sets that involve millions of data points.

The remainder of the manuscript is structured as follows: Section 2 presents a brief overview of the MPR model and parameter estimation; more details are given in Žuković and Hristopulos, 2017). In Section 3 we present the CUDA implementation of the MPR model. The statistical and computational performance of the model is investigated in Section 4. Finally, in Section 5 we present a summary and conclusions.

2 MPR Method

An efficient and automatic simulation method (hereafter called the MPR method), has been recently introduced for the prediction of non-Gaussian data partially sampled on Cartesian grids (Žuković and Hristopulos, 2017). The MPR method is based on a Gibbs-Markov random field (GMRF) that employs the modified planar rotator (MPR) model. The idea of the
MPR method is to transform the original data to continuously-valued “spin” variables, and then capture spatial correlations via short-range interactions between the spins using a modified version of the well known planar rotator model from statistical physics. This approach can account for spatial correlations that are typical in geophysical and environmental data sets. The energy functional $H$ of the MPR model measures the “cost” of each spatial configuration: higher-cost configurations have a lower probability of occurrence than lower-cost ones. The MPR energy function is given by

$$H = -J \sum_{\langle i,j \rangle} \cos[q(\phi_i - \phi_j)],$$  \hspace{1cm} (1)

where $J > 0$ is the exchange interaction parameter, $\langle i,j \rangle$ denotes the sum over nearest neighbor spins on the grid, and $q \leq 1/2$ is the modification factor. The joint probability density function of the GMRF is then given by

$$f = \frac{1}{Z} \exp(-H/k_BT),$$  \hspace{1cm} (2)

where the normalization constant $Z$ is the partition function, $k_B$ is the Boltzmann constant, and $T$ is the temperature parameter (higher temperature favors larger fluctuation variance).

The spatial prediction at missing data sites is based on performing conditional Monte Carlo (MC) simulations and taking the mean of the respective conditional distribution. In thermodynamic equilibrium, the MPR model has been shown to display a flexible correlation structure controlled by the temperature. The latter is the only model parameter, and it is given in dimensionless units as it absorbs both the exchange interaction parameter and the Boltzmann constant (Zukovič and Hristopulos, 2015). Assuming ergodic conditions, the temperature is automatically and efficiently estimated based on an ergodic specific energy matching principle, which is analogous to the classical statistical method of moments.
Algorithm 1 Simplified gap-filling algorithm based on conditional simulation with the MPR model.

1: Transformation of original sample data $Z_s \in [Z_{min}, Z_{max}]$ to the spin representation $\Phi_s \in [0, 2\pi]$
2: Parameter ($T$) estimation using specific energy matching principle
3: Initialization of spin state and setting simulation parameters
4: Non-equilibrium spin relaxation using efficient hybrid (restricted Metropolis + over-relaxation) MC simulation with automatic detection of equilibrium onset
5: Equilibrium state simulation conditionally on the sample values
6: Inverse transformation from spin $\Phi$ to original $Z$ values
7: Prediction based on the conditional mean of simulation ensemble
8: Calculation of simulation statistics and evaluation of predictive performance

A vectorized hybrid MC simulation algorithm leads to fast relaxation followed by equilibrium simulation. The total computational time scales approximately linearly with the system size. The demonstrated efficiency of the MPR method makes it suitable for big data sets, such as satellite and radar images. The MPR algorithm is described in detail in Žuković and Hristopulos (2017). The basic steps are summarized in Algorithm 1.

3 CUDA Programming Model and its Implementation

CUDA (Compute Unified Device Architecture) is a general purpose parallel computing platform and programming model created by NVIDIA and implemented by the NVIDIA GPUs, which leverages the power of parallel computing on GPUs to solve complex computational problems more efficiently than on a single CPU. CUDA comes with a software environment that extends the capabilities of C programming language, thus allowing developers to create their own parallelized applications. More details of the CUDA architecture are presented in Nvidia (2018). Below, we give a brief overview of the CUDA programming model and highlight the features that are critical for the improved performance of the MPR method.

CUDA C code is essentially C code with additional syntax, keywords and API functions. Based on those the CUDA compiler (nvcc) splits the code into multiple parts, which are ex-
executed on either the CPU, which is known as the host (consequently, this part of the code is called host code), or the GPU, which is known as the device (hence, this part is known as the device code). The most basic concept of a CUDA program is as follows: Some data is stored in the host memory (i.e. RAM) representing input data for computations. First, the host copies the data to the memory allocated on the device and then the device performs computations using device functions called kernels. The latter harness the parallel computing capabilities of the GPU, solving the problem in an efficient and optimized manner. Finally, the device copies the results back to the host memory. In the following paragraphs we review the implementation of parallel computing by programming kernels and explain the differences between various types of memory available on the GPU. The proper setup of the CUDA programming model is crucial for optimal computational performance.

3.1 CUDA Kernels

A GPU kernel is a C/C++ function written in a serial way that can be executed on many GPU threads in parallel. A GPU kernel is run on blocks of threads (very simple software units - virtual processors - that can be mapped to hardware resources to execute computations) organized into a grid. The grid blocks and the number of threads per block are specified at launch. The blocks of threads are scheduled to run on the GPU cores organized into streaming multiprocessors (SMs). Each thread can execute the same kernel function on different input data points. This is called SIMD (Single Instruction Multiple Data) approach and implements data-level parallelism. Instruction level parallelism is achieved by launching multiple kernels in parallel. Each thread in a block, as well as each block in the grid, has a unique index, which results in a well-defined unique index for each thread. Using this

1 Note that the term grid in this subsection refers to a unit in the GPU architecture and should not be confused with the spatial grids used in the remainder of the paper.
thread index, we can specify which piece of data to process and we can control a decision flow for any thread. This architecture is illustrated in the schematic of Fig. 1.

It is generally good practice to run as many threads as there are pieces of data that can be processed in parallel. For example, to perform a computation on an array of values of length $N$, we would launch $N$ threads. For matrix operations, e.g., matrix multiplication of two $N \times N$ matrices, we would execute, hardware permitting, $N \times N$ threads. Threads in a block as well as blocks in a grid can be arranged in one, two or three dimensions. The hardware limits the number of threads per block to a maximum of 1024 (512 on older devices with compute capability less than 2.0).

Even the latest graphics cards contain at most 5120 computational cores, which means that the number of “software threads” may exceed that of hardware cores by many orders of magnitude. This means that possibly not all threads can run concurrently on the GPU; some threads may have to be scheduled after previous threads finish executing. Threads are scheduled to run on an SM in blocks, and a single SM can execute one or more thread blocks. Individual threads are executed in groups called warps. Warps get scheduled to run

---

2 The compute capability of a GPU determines its general specifications and available features, i.e., maximum size of a block, how many blocks fit on a multiprocessor or whether or not is the GPU capable of certain operations.
on an SM automatically by the warp scheduler, which can also stall warps waiting for data or memory transactions and schedule others in their place, so that as many warps as possible run concurrently. If SMs are occupied with a sufficiently large number of warps, this scheduler mechanism can effectively hide the high latency of memory operations, which is crucial for a highly performing CUDA application.

3.2 Memory Hierarchy

There are several types of memory available on the GPU, each with its own uses, advantages, and disadvantages. We will review the three most important ones here. A visual representation of the memory hierarchy is shown in Fig. 2.

3.2.1 Global Memory

Global memory (also called device memory or DRAM) is located on the graphics card but off the GPU chip itself. Global memory can be accessed by any active thread in the whole grid; thus, it can be used for communication between any given threads. It is by far the largest (up to 48 GB on latest professional cards, 4 GB on GTX 980 used in this work) but also the slowest, as it takes about 500 clock cycles to read or write data, compared to one clock cycle.
for an arithmetic operation. This is however still orders of magnitude faster than writing or reading directly from the host memory. Consequently, for performance reasons it is vital to maximize arithmetic intensity, i.e., the number of arithmetic operations per read/write cycle.

Another vital optimization strategy is called coalesced memory access. When a thread requests data stored in the global memory, a memory transaction is executed. This transaction, however, fetches memory data in chunks much larger than a single thread request. If adjacent threads in a warp can also benefit from this transaction by having all of their data fetched, memory bandwidth is not wasted and the memory transaction is considered coalesced. The device coalesces the global memory loads and stores, which are issued by threads of a warp, into as few transactions as possible to minimize DRAM bandwidth (Harris, 2013). In practice, one simply has to ensure, if possible, that threads with adjacent indices access adjacent memory locations. Memory coalescing is vital for both reading from and writing to the global memory.

3.2.2 Shared Memory

Shared memory is located on the GPU chip and provides an option for low latency communication between threads within a block, since reading or writing to it is much faster than using the global memory. The size of shared memory is, however severely limited, for instance at 48 KB per thread block and 96 KB per SM on our GTX 980. A common strategy for utilizing shared memory is as follows: First, store the data needed for the block in its shared memory, then perform computations requiring reading/writing only on the shared data, and finally output the block results to the global memory for further use. This approach minimizes high latency global memory access and it is desirable in reduction algorithms, such as the calculation of a sum of data array.
3.2.3 Registers

The per thread local memory is implemented by 32 bit (4 byte) registers and is used for storing local variables of a particular thread. Registers provide extremely low latency memory access. In past GPU architectures (before Kepler) this per thread local memory was only accessible by the same thread. However, with the advent of the Kepler architecture (2012), threads belonging to the same warp can read each other’s registers using a warp shuffle operation. The number of registers used per block is capped to a maximum of 65536.

3.2.4 Other memories

For completeness, there are two more types of memory - texture and constant. They both reside in DRAM, are read-only and cached. If properly used they should give performance benefit over global memory reads. The constant memory is optimized for broadcasting the same value to all threads in warp and the texture memory is more suitable for data with 2D spatial locality.

3.3 Implementation of the MPR Method

Achieving maximum speedup of the MPR method hinges on efficient (parallel) implementation of the hybrid algorithm that combines deterministic over-relaxation with a stochastic Metropolis step. Our approach is based on the parallel implementation of the Metropolis algorithm for the Ising model (Weigel, 2011, 2012).

In the following we assume that the data are supported on a square, two-dimensional grid (extension to rectangular grids is straightforward). Square grids are bipartite graphs, meaning that they can be divided into two disjoint and independent graphs (sub-grids), e.g., A and B. Most importantly, the nearest neighbors of any node on the sub-grid A belong to
Fig. 3 Checkerboard decomposition of the spin grid. Each square represents a grid node with its associated spin variable. Sub-grid A consists of the dark nodes, while sub-grid B comprises the light nodes. The green (shaded) tile in the upper left corner represents a block of threads that are responsible for the numerical operations on the spins included within the square. Each thread performs calculations for a single spin.

the sub-grid B, and vice versa. Hence, the MPR model’s set of variables (“spins”) can be split between two sub-grids, so that spins in the first sub-grid only interact with spins of the second sub-grid and vice versa. This is called the checkerboard decomposition, and it is depicted in Fig. 3. By means of this decomposition it is possible to apply the updating algorithm to all the sub-grid spins in parallel. To perform the necessary calculations, one thread is called per each sub-grid spin.

Let us recall that the used hybrid MC updating scheme involves two steps: the deterministic over-relaxation update that conserves the energy is combined with the stochastic Metropolis update to achieve ergodicity. In the case of the over-relaxation step, each thread calculates the local energy contribution of its assigned spin on the first sub-grid, and subsequently updates the spin value so that the local energy contribution remains unchanged. In the case of the Metropolis step, each thread loads both the old and new spin states into its registers, calculates the local energy difference between the states, and attempts to update the spin according to the Metropolis criterion. Once these steps are completed, the same procedure is also applied to the second sub-grid.
To avoid any unnecessary overhead, it is important to generate the random numbers needed for proposing new states and the Metropolis updates beforehand and store them in the GPUs global memory. For each spin and each MC step two random numbers are needed — one to propose a new state for the given spin and another one to stochastically accept it. They were generated by Philox4_32_10 random number generator from cuRAND library.

3.3.1 Launch Configuration

The kernel launch configuration for the spin updating procedure is an important aspect of the implementation. It involves various settings, such as specifying grid and block dimensions and the amount of shared memory that the kernel expects to use. We need as many threads as there are spins in each sub-grid arranged into a grid of thread blocks in a way that keeps the index arithmetic as simple as possible, yet also coalesces global memory access.

For our two-dimensional (2D) system it is natural to use a 2D grid of 2D blocks (squares). The block size, however, should be carefully considered as it can affect hardware utilization, which is measured by a metric called occupancy (Nvidia, 2018). Occupancy is the ratio of the number of active warps per SM to the maximum number of possible active warps. Two factors limit how many warps can be run on a SM in parallel: the maximum number of blocks per SM, which depends on the compute capability of the device and can be found in its technical specifications, and the availability of resources such as the shared memory and registers. The block size is limited to 1024 threads per block on devices with at least 2.0 compute capability. The optimal size of a (square) block should always be a multiple of warp size (32) and, thus, considering the limitation of 1024 threads per block, the candidate sizes are 64, 256, and 1024. Performance based on different block sizes (parametrized by their side length \( B \)) is presented in Section 4.
3.3.2 Effects of Arithmetic Precision

The numerical precision of the calculations affects the execution time on the GPU considerably more than on the CPU. If the problem permits, one can achieve remarkable speedups by switching from double to single precision, and by making use of fast, precision-specific CUDA math functions. For example, if we opt for single precision, instead of functions such as \( \sin() \), \( \cos() \) or \( \exp() \) we can use their single precision counterparts, i.e., \( \sinf() \), \( \cosf() \) and \( \expf() \), which are significantly faster than their more precise analogues. However, it should be ensured that using lower precision does not severely affect the results due to excessive rounding errors.

4 Results

The performance of the MPR method, implemented on CPU in the C++ and on GPU in the CUDA environments, is validated using synthetically generated spatial data. For the sake of consistency, we generate data with the same statistical properties as in Žuković and Hristopulos (2017); therein the data were processed on CPU in the Matlab\textsuperscript{®} environment. Namely, samples from the Gaussian random field \( Z \sim N(m = 50, \sigma = 10) \) with Whittle-Matérn (WM) covariance are simulated on square grids with \( L \) nodes per side for different values of \( L \). The covariance function is given by

\[
G_Z(\|h\|) = \frac{2^{1-\nu} \sigma^2}{\Gamma(\nu)} (\kappa \|h\|)^\nu K_\nu(\kappa \|h\|),
\]

(3)

where \( \|h\| \) is the Euclidean two-point distance, \( \sigma^2 \) is the variance, \( \nu \) is the smoothness parameter, \( \kappa \) is the inverse autocorrelation length, and \( K_\nu \) is the modified Bessel function of index \( \nu \). The simulations are based on the spectral method (Drummond and Horgan, 1987).
From the simulated data, we generate \( S = 100 \) different sampling configurations by random removal of \( p = 33\% \) and \( 66\% \) of points. The MPR predictions at the removed points are calculated and compared with the true values using different validation measures. For consistency with the Matlab\textsuperscript{®} results reported by \cite{Zukovic2017}, we evaluate the same validation measures, i.e., the mean average absolute error (MAAE), the mean average relative error (MARE), the mean average absolute relative error (MAARE), and the mean root average squared error (MRASE). For both the CPU and the GPU calculations we measure the respective run times \( \langle t_{\text{cpu}} \rangle \) and \( \langle t_{\text{gpu}} \rangle \) and evaluate the speedup of the CPU Matlab\textsuperscript{®} as well as GPU CUDA implementations relative to the single-CPU C++ implementation. The calculations were conducted on a desktop computer with 16.0 GB RAM, Intel\textsuperscript{®} Core\textsuperscript{TM} i5-4440 CPU processor with an 3.1 GHz clock (up to 3.3 GHz Max Turbo frequency) and NVIDIA GeForce GTX 980 GPU with 2048 CUDA cores and 1178 MHz clock (up to 1279 MHz Boost frequency).

In Table 1 we present validation measures obtained with double-precision (DP) arithmetic for random fields with WM covariance parameters \( \nu = 0.5, \kappa = 0.2 \) on a square grid with \( L = 1024 \) in the three different programming environments. There are only minute differences between the validation measures obtained by the Matlab\textsuperscript{®}, C++ and CUDA implementations, which are most likely caused by using different sequences of random numbers. Similar results are also obtained for other values of the WM parameters. Therefore, in the following we set \( \nu = 0.5, \kappa = 0.2 \) and focus on other factors that affect computational efficiency. In particular, the choice of the programming environment can strongly affect the computational efficiency. The bottom row of Table 1 shows that, in spite of the vectorization based on the checkerboard algorithm the Matlab\textsuperscript{®} code is about 7 – 9\% slower than the C++ code. This is not surprising, as the former is well known to be less efficient, particularly if the algorithm involves a large number of iterations. On the other hand, the GPU imple-
Table 1 Validation measures obtained in different programming environments for random field data with WM covariance parameters $\nu = 0.5, \kappa = 0.2$. The data are generated on a square grid with length $L = 1024$ for different values of sparsity, and the calculations use double precision arithmetic.

|                | CPU (C++) | CPU (MATLAB) | GPU ($B = 16$) |
|----------------|-----------|--------------|----------------|
|                | $p = 33\%$ | $p = 66\%$  | $p = 33\%$ | $p = 66\%$ | $p = 33\%$ | $p = 66\%$ |
| MAAE           | 3.381     | 3.828        | 3.379        | 3.828        | 3.380        | 3.827        |
| MARE [\%]     | $-0.989$   | $-1.303$     | $-0.993$     | $-1.299$     | $-0.991$     | $-1.297$     |
| MAARE [\%]    | 7.172     | 8.157        | 7.171        | 8.157        | 7.172        | 8.156        |
| MRASE          | 4.245     | 4.821        | 4.247        | 4.822        | 4.244        | 4.821        |
| $\langle t_{dp} \rangle$ [s] | 38.665   | 69.953       | 42.484       | 75.092       | 0.893        | 0.883        |
| speedup        | 1.000     | 1.000        | 0.910        | 0.932        | 43.279       | 79.200       |

Implementation leads to a dramatic increase of speed. More specifically, for the chosen block side length $B = 16$ and data sparsity $p = 33\%$, the CUDA GPU time is 43 times smaller than the C++ CPU time and the speedup increases with $p$ up to almost 80 times for $p = 66\%$.

Further appreciable gains in speed without noticeable compromise of the predictive performance can be obtained by opting for single-precision (SP) calculations. Table 2 shows that for the same parameters as in Table 1 the change to single precision increases the speedup by almost four times, i.e., to 166 for $p = 33\%$ and up to 306 for $p = 66\%$, while all the validations measures remain the same (up to at least the third decimal place). Table 2 also demonstrates the effect of the block size selection. The value of the block length $B$ does not seem to affect the prediction performance, but it can be optimized with respect to speed.

Based on our test results, optimal speed is achieved for $B = 16$. Choosing other block values (particularly smaller, i.e., $B = 8$) leads to less efficient performance. This is in agreement with the findings reported by Weigel (2011, 2012).

The results presented in the above tables pertain to fixed grid side length $L = 1024$. Targeting massive data sets, it is interesting to investigate the dependence of computational time on increasing grid size. Fig. 4 illustrates the dependence of the computational time, on both CPU and GPU, as well as for different sparsity values $p$. Figs. 4(a) and 4(b) show results for the double- and single-precision calculations, respectively. For the CPU data, the
Table 2 Validation measures obtained in different programming environments for random field data with WM covariance parameters $\nu = 0.5, \kappa = 0.2$. The data are generated on a square grid with length $L = 1024$ for different values of sparsity, and the calculations use single precision arithmetic.

|            | CPU (C++) | GPU ($B = 8$) | GPU ($B = 16$) | GPU ($B = 32$) |
|------------|-----------|---------------|----------------|----------------|
| $p = 33\%$ | 3.380     | 3.380         | 3.380          | 3.380          |
| $p = 66\%$ | 3.828     | 3.827         | 3.827          | 3.827          |
| MAAE       | -0.992    | -1.300        | -1.297         | -1.297         |
| MARE [%]   | 7.172     | 8.158         | 8.156          | 8.156          |
| MAARE [%]  | 4.244     | 4.821         | 4.821          | 4.821          |
| $\langle t_{SP} \rangle$ [s] | 26.899 | 52.771 | 0.207 | 0.223 |
| speedup    | 1.000     | 1.000         | 129.972        | 236.575        |

Fig. 4 Comparison of the computational times (in seconds) required by CPU (circles) and GPU (squares) calculations and their dependence on the grid length $L$, for $p = 33\%$ (green) and $p = 66\%$ (red). Panels (a) and (b) show results for double and single precision arithmetic, respectively. The dashed line in (b) shows the linear fit to the grid size $L^2$. The GPU times are little sensitive to the data sparsity (green and red squares almost coincide).

Fig. 5 Ratio of double- over the single-precision computational times obtained on CPU (circles) and GPU (squares), for $p = 33\%$ (green) and 66\% (red) versus the grid length $L$. 
increase of the computational time follows an approximately linear dependence at all scales, in line with the Matlab® checkerboard results in Žuković and Hristopulos (2017). The CPU times range between a few milliseconds for the smallest, \( L = 32 \), up to about one hour for the largest, \( L = 8192 \) grid lengths. Naturally, the single-precision calculations are faster than the respective double-precision ones. However, the ratio of the computational times does not seem to vary systematically with \( L \) and does not exceed 44%.

On the other hand, there appear to be two regimes for the GPU times. For small to moderate \( L \) (\( L \sim 10^2 \sim 10^3 \)) the increase with size is very gentle due to the low SMs occupancy that cannot completely hide memory latency. Only for larger grid sizes, when SMs are fully utilized, the GPU times follow about the same linear increase as the CPU ones (see the dashed line in Fig. 4(b)). In contrast with the CPU calculations, the relative difference between the SP-GPU and DP-GPU times systematically increases for \( L > 68 \); for \( L \) exceeding 1024 the SP-GPU execution is faster than the DP-GPU, even by as much as five times, as shown in Fig. 5. Consequently, the SP-GPU times range between a few milliseconds for the smallest \( L = 32 \) (for small \( L \) they are even slightly larger than the CPU times) up to about nine seconds for the largest \( L = 8192 \). Note that the DP-GPU calculations could not be performed for the largest, \( L = 8192 \), grid length due to insufficient global memory.

Finally, we investigate the impact of the sample’s sparsity on computational speed. Generally, higher sparsity means a higher number of prediction points that enter conditional Monte Carlo simulations, and therefore larger computational demands. In Fig. 6(a) we show that the CPU time increases roughly linearly with sparsity \( p \), for both DP and SP calculations. On the other hand, the GPU times presented in Fig. 6(b) display a more complex behavior with increasing \( p \). Namely, a steeper increase for smaller \( p \) is followed by a flatter part within \( 20\% \lesssim p \lesssim 70\% \) which is then followed by another steep increase for \( p \gtrsim 70\% \).
Fig. 6 (a) CPU and (b) GPU times as functions of the sample sparsity $p$, obtained by DP (circles) and SP (squares) calculations for grid length $L = 2048$.

Again, the ratio of the DP-GPU over the SP-GPU times, i.e., $\langle t_{dp}^{cpu} \rangle / \langle t_{sp}^{cpu} \rangle \approx 5.1$, significantly exceeds the value of the ratio, i.e., $\langle t_{dp}^{gpu} \rangle / \langle t_{sp}^{gpu} \rangle \approx 1.1$, for the CPU calculations.

Fig. 7(a) shows the speedup achieved by the GPU CUDA implementation compared to the single-CPU C++ implementation as a function of the grid length, for SP (circles) and DP (squares) calculations and two values of the thinning $p = 33\%$ (green symbols) and $66\%$ (red symbols). As is evident, for the smallest grid length $L = 32$ the GPU implementation has no advantage over the CPU implementation in terms of computational speed. In fact, for $p = 33\%$ the GPU run time is even larger than the CPU time (speedup is less than one). Nevertheless, the speedup factor dramatically increases with the grid size, and the full potential of the GPU code shows up for lengths $L \gtrsim 1024$, at which all the speedup curves appear to level off. The speedup achieved by SP calculations increases with the grid length faster than the speedup for DP calculations. For example, for $L = 32$ the speedup of the DP calculations is 0.61 for $p = 33\%$ and 1.01 for $p = 66\%$ versus 0.63 for $p = 33\%$ and 1.06 for $p = 66\%$ recorded for the SP calculations. On the other hand, for $L = 2048$ the DP values are 49.92 for $p = 33\%$ and 83.95 for $p = 66\%$ versus the SP values of 251.50 for $p = 33\%$ and 392.10 for $p = 66\%$. 
Fig. 7 Speedups of the GPU-accelerated MPR method as functions of (a) the grid length $L$ for data with $p = 33\%$ (green) and $p = 66\%$ (red) and (b) the sparsity $p$ for $L = 2048$, obtained from SP (circles) and DP (squares) calculations.

The speedup increase with the sample sparsity $p$ is illustrated in Fig. 7(b). The speedup increases for both the DP and SP schemes, but the magnitude of the latter is more than five times larger for every $p$. In particular, for very sparse samples the SP calculations on GPU can be almost 500 times faster than those performed on CPU.

5 Summary and Conclusion

The ever-increasing amount of spatial data calls for new computationally efficient methods and for optimal use of available computational resources. Recently, a novel spatial prediction method (MPR), inspired from statistical physics, was introduced for the reconstruction of missing data on regular grids. In spite of its simplicity, the MPR method was shown to be computationally efficient and competitive with several interpolation methods (Žukovič and Hristopulos, 2017). The MPR efficiency derives from the local nature of the interactions between the MPR model’s variables and from an efficient hybrid simulation algorithm. Thus, the computational time of the MPR method scales approximately linearly with system size. The computational speed along with the ability for automatic operation make the MPR method promising for near real-time processing of massive raster data. Fur-
In the present paper, we take advantage of the local (nearest-neighbor) interactions of the MPR model to provide a parallel implementation on general purpose GPUs in the CUDA environment. To demonstrate the computational speedup achievable by the GPU implementation, we perform tests on synthetic data sets with randomly missing values. The data sets have different sizes and sparsity. The tests are run in both CUDA on GPU and C++ on a single CPU. We first show that the speedup can be optimized by a thoughtful setting the GPU environment, such as the block size. For the range of grid sizes in this study, the value of the block side length $B = 16$ is found to be optimal.

In line with our earlier results (ˇZukoviˇc and Hristopoulos, 2017), the CPU time is confirmed to increase approximately linearly with the grid size $L^2$ over the entire range of $L$. On the other hand, the increase of the GPU time with grid length is initially very gentle until the linear regime is established for $L \gtrsim 2048$. Another advantage of the GPU over the CPU implementation is opting for single- instead of double-precision calculations. While there is no significant gain in single over double precision on CPU, in the GPU implementation for large enough $L$ the speedup can be more than fivefold with no observable deterioration of the prediction performance. The speedup is also found to increase with the sample sparsity. For very sparse data on large grids, the speedup due to single precision arithmetic can be as large as almost 500 times. Thus, using an ordinary personal computer, data sets that involve up to hundreds of thousands of points with arbitrary sparsity can be processed in almost real time and data sets that involve millions of points can be processed in less than one second.

**Acknowledgements** This work was supported by the Scientific Grant Agency of Ministry of Education of Slovak Republic (Grant Nos. 1/0474/16 and 1/0331/15). We also acknowledge support for a short visit by...
M. Ž. at the Technical University of Crete from the Hellenic Ministry of Education - Department of Inter-University Relations, the State Scholarships Foundation of Greece and the Slovak Republic’s Ministry of Education through the Bilateral Programme of Educational Exchanges between Greece and Slovakia.

References

Cheng T (2013) Accelerating universal kriging interpolation algorithm using CUDA-enabled GPU. Computers & Geosciences 54:178 – 183

Cheng T, Li D, Wang Q (2010) On parallelizing universal kriging interpolation based on OpenMP. In 2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science, 36–39

Cressie N, Johannesson G (2018) Fixed rank kriging for very large spatial data sets. Journal of the Royal Statistical Society: Series B (Statistical Methodology) 70(1):209–226

de Rav E G, Jimnez-Hornero F, Ariza-Villaverde A, Gmez-Lpez J (2014) Using general-purpose computing on graphics processing units (GPGPU) to accelerate the ordinary kriging algorithm. Computers & Geosciences 64:1–6

Drummond I T, Horgan R R (1987) The effective permeability of a random medium. Journal of Physics A: Mathematical and General 20(14):4661

Furrer R, Genton M G, Nychka D (2006) Covariance tapering for interpolation of large spatial datasets. Journal of Computational and Graphical Statistics 15(3):502–523

Guan Q, Kyrakiakis P C, Goodchild M F (2011) A parallel computing approach to fast geostatistical areal interpolation. International Journal of Geographical Information Science 25(8):1241–1267

Harris M (2013) How to Access Global Memory Efficiently in CUDA C/C++ Kernels

Hartman L, Hossjer O (2008) Fast kriging of large data sets with Gaussian Markov random fields. Computational Statistics & Data Analysis 52(5):2331 – 2349, ISSN 0167-9473

Hristopulos D (2003) Spartan Gibbs random field models for geostatistical applications. SIAM Journal on Scientific Computing 24(6):2125–2162

Hristopulos D T (2015) Stochastic local interaction (sli) model. Computers & Geosciences 85(PB):26–37

Hristopulos D T, Eloge S N (2007) Analytic properties and covariance functions for a new class of generalized Gibbs random fields. IEEE Transactions on Information Theory 53(12):4667–4679
Hu H, Shu H (2015) An improved coarse-grained parallel algorithm for computational acceleration of ordinary kriging interpolation. Computers & Geosciences 78:44 – 52

Ingram B, Cornford D, Evans D (2008) Fast algorithms for automatic mapping with space-limited covariance functions. Stochastic Environmental Research and Risk Assessment 22(5):661–670

Kaufman C G, Schervish M J, Nychka D W (2008) Covariance tapering for likelihood-based estimation in large spatial data sets. Journal of the American Statistical Association 103(484):1545–1555

Kerry K E, Hawick K A (1998) Kriging interpolation on high-performance computers. In Sloot P, Bubak M, Hertzberger B, editors, High-Performance Computing and Networking, Berlin, Heidelberg: Springer Berlin Heidelberg, 429–438

Marcotte D, Allard D (2018) Half-tapering strategy for conditional simulation with large datasets. Stochastic Environmental Research and Risk Assessment 32(1):279–294

Mei G (2014) Evaluating the power of GPU acceleration for IDW interpolation algorithm. The Scientific World Journal 2014:1715741 – 8

Mei G, Xu L, Xu N (2017) Accelerating adaptive inverse distance weighting interpolation algorithm on a graphics processing unit. Open Science 4(9)

Nvidia (2018) Cuda C Best Practices Guide

Nvidia (2018) CUDA C Programming Guide, version 10.0

Pesquer L, Corts A, Pons X (2011) Parallel ordinary kriging interpolation incorporating automatic variogram fitting. Computers & Geosciences 37(4):464 – 473, ISSN 0098-3004

Sickles J E, Shadwick D S (2007) Effects of missing seasonal data on estimates of period means of dry and wet deposition. Atmospheric Environment 41(23):4931 – 4939, ISSN 1352-2310

Stojanovic N, Stojanovic D (2014) High performance processing and analysis of geospatial data using CUDA on GPU. Advances in Electrical and Computer Engineering 14(4):109–114

Tahmasebi P, Sahimi M, Mariethoz G, Hezarkhani A (2012) Accelerating geostatistical simulations using graphics processing units (GPU). Computers & Geosciences 46:51 – 59

Wackernagel H (2003) Multivariate Geostatistics. Springer-Verlag Berlin Heidelberg, 3rd edition

Weigel M (2011) Simulating spin models on gpus. Computer Physics Communications 182(9):1833 – 1836, ISSN 0010-4655, computer Physics Communications Special Edition for Conference on Computational Physics Trondheim, Norway, June 23-26, 2010
Weigel M (2012) Performance potential for simulating spin models on GPU. Journal of Computational Physics 231(8):3064 – 3082
Xia Y J, Kuang L, Li X M (2011) Accelerating geospatial analysis on GPUs using CUDA. Journal of Zhejiang University, SCIENCE C 12(12):990–999
Zhang W, Li W, Zhang C, Zhao T (2018a) Parallel computing solutions for Markov chain spatial sequential simulation of categorical fields. International Journal of Digital Earth 0(0):1–17
Zhang Y, Zheng X, Wang Z, Ai G, Huang Q (2018b) Implementation of a parallel gpu-based space-time kriging framework. ISPRS International Journal of Geo-Information 7(5), ISSN 2220-9964
Zhong X, Kealy A, Duckham M (2016) Stream kriging: Incremental and recursive ordinary kriging over spatiotemporal data streams. Computers & Geosciences 90:134 – 143
Žukovič M, Hristopulos D T (2009a) Classification of missing values in spatial data using spin models. Physical Review E 80:011116
Žukovič M, Hristopulos D T (2009b) Multilevel discretized random field models with 'spin' correlations for the simulation of environmental spatial data. Journal of Statistical Mechanics: Theory and Experiment 2009(02):P02023
Žukovič M, Hristopulos D T (2013a) A directional gradient-curvature method for gap filling of gridded environmental spatial data with potentially anisotropic correlations. Atmospheric Environment 77:901–909
Žukovič M, Hristopulos D T (2013b) Reconstruction of missing data in remote sensing images using conditional stochastic optimization with global geometric constraints. Stochastic Environmental Research and Risk Assessment 27(4):785–806
Žukovič M, Hristopulos D T (2015) Short-range correlations in modified planar rotator model. Journal of Physics: Conference Series 633(1):012105
Žukovič M, Hristopulos D T (2017) Gibbs Markov Random Fields with Continuous Values based on the Modified Planar Rotator Model. arXiv preprint arXiv:171003038